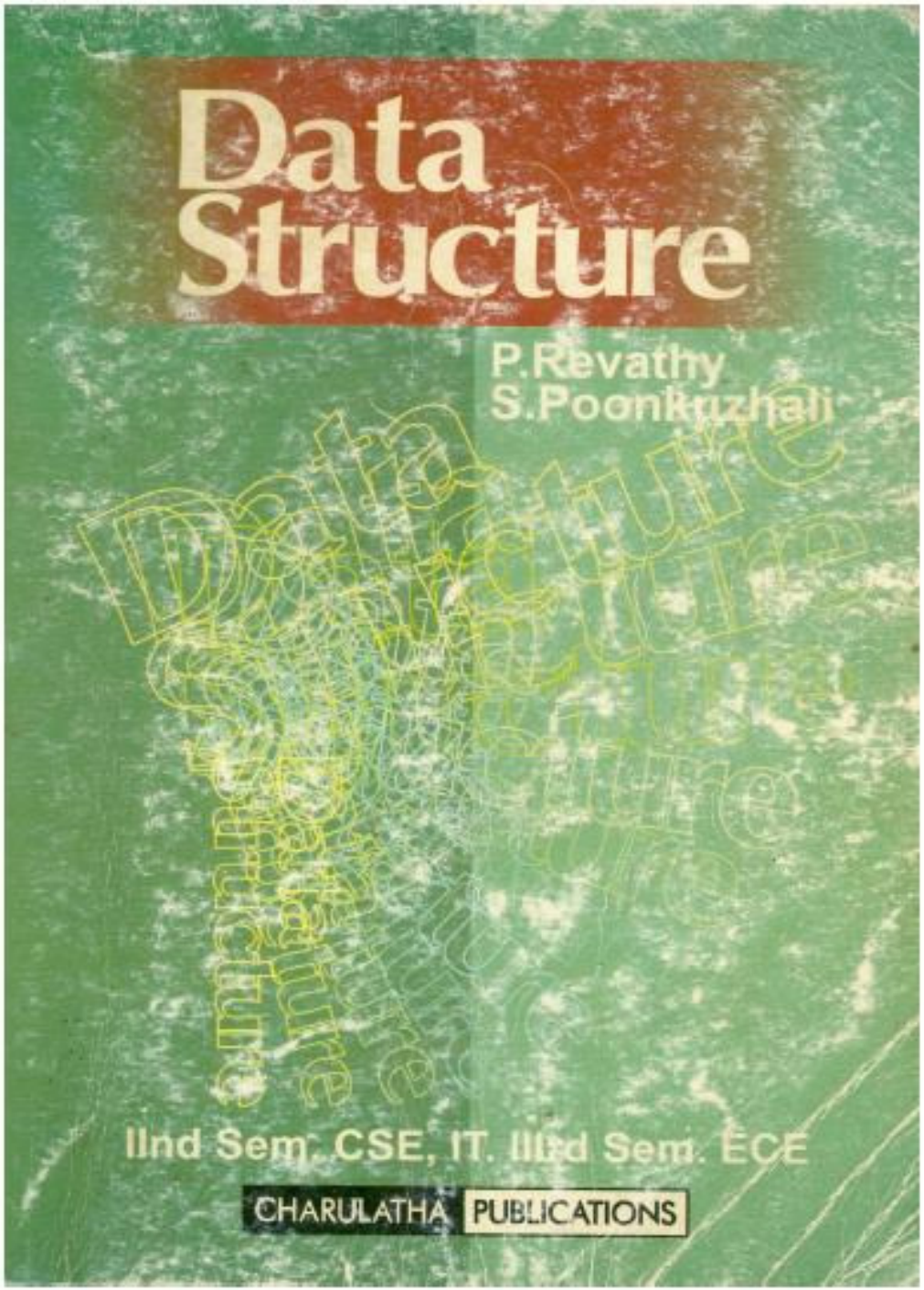


DATA STRUCTURES

Designed as per

ANNA UNIVERSITY Syllabus

For

B.E., IISem. CSE,IT & III Sem. ECE

P. REVATHY

S. POONKUZHALI

Department of CSE

RAJALAKSHIMI ENGINEERING COLLEGE

CHENNAI

CHARULATHA PUBLICATIONS

New No. 24, Thambiah Road

West Mambalam, Chennai - 600 033.

Phone : 24745589, 24746546

Email : charulathapublications@yahoo.com

Acknowledgement

We take this opportunity to express our deep gratitude to our Correspondent ***Thiru. S.Meganathan***, chairperson ***Dr. Thangam Meganathan***, Principal ***Dr K. Sarukesi***, Dean ***Prof. V. N.Srinivasan*** and Head of the Department of Computer Science ***Ms. S. Pramila*** for their inspiration and encouragement in bringing out this book successfully.

Many of our colleagues have read various portions of the manuscript and have given us valuable comments and advice. In particular we would like to thank ***Ms. J. Ida Christy***, ***Ms. N. Sunitha*** and ***Ms. Dhalphena Phemila***.

Of course, we would have nothing to write about without the many people who did their original research that provided the material we enjoy learning and passing on to new generation of students. We thank them for their work.

We would like to extend our thanks to ***Mr. M.R. Bharathi*** of Charulatha Publications.

We would like to give our warmest thanks to ***Mr. N. Mahesh Kumar*** for his expert assistance in helping to prepare the manuscript for typesetting and Charulatha Publications in bringing out this book.

Finally, we would like to thank our family members who had been a constant source of moral support throughout this incredibly project.

Suggestions and comments for further improvement of this book are most welcome. You may please email us at

prevathy@hotmail.com

kuzhal_s@yahoo.co.in

P. Revathy

S. Poonkuzhali

CS1151 - DATA STRUCTURES

AIM

To provide an in-depth knowledge in problem solving techniques and data structures.

OBJECTIVES

- To learn the systematic way of solving problems
- To understand the different methods of organizing large amounts of data
- To learn to program in C
- To efficiently implement the different data structures
- To efficiently implement solutions for specific problems

UNIT I - PROBLEM SOLVING

Problem solving – Top-down Design – Implementation – Verification – Efficiency – Analysis – Sample algorithms.

UNIT II - LISTS, STACKS AND QUEUES

Abstract Data Type (ADT) – The List ADT – The Stack ADT – The Queue ADT

UNIT III - TREES

Preliminaries – Binary Trees – The Search Tree ADT – Binary Search Trees – AVL Trees – Tree Traversals – Hashing – General Idea – Hash Function – Separate Chaining – Open Addressing – Linear Probing – Priority Queues (Heaps) – Model – Simple implementations – Binary Heap

UNIT IV - SORTING

Preliminaries – Insertion Sort – Shellsort – Heapsort – Mergesort – Quicksort – External Sorting

UNIT V - GRAPHS

Definitions – Topological Sort – Shortest-Path Algorithms – Unweighted Shortest Paths – Dijkstra's Algorithm – Minimum Spanning Tree – Prim's Algorithm – Applications of Depth-First Search – Undirected Graphs – Biconnectivity – Introduction to NP-Completeness

TUTORIAL

TEXT BOOKS

1. R. G. Dromey, "How to Solve it by Computer" (Chaps 1-2), Prentice-Hall of India, 2002.
2. M. A. Weiss, "Data Structures and Algorithm Analysis in C", 2nd ed, Pearson Education Asia, 2002. (chaps 3, 4.1-4.4 (except 4.3.6), 4.6, 5.1-5.4.1, 6.1-6.3.3, 7.1-7.7 (except 7.2.2, 7.4.1, 7.5.1, 7.6.1, 7.7.5, 7.7.6), 7.11, 9.1-9.3.2, 9.5-9.5.1, 9.6-9.6.2, 9.7)

REFERENCES

1. Y. Langsam, M. J. Augenstein and A. M. Tenenbaum, "Data Structures using C", Pearson Education Asia, 2004
2. Richard F. Gilberg, Behrouz A. Forouzan, "Data Structures – A Pseudocode Approach with C", Thomson Brooks / COLE, 1998.
3. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms", Pearson education Asia, 1983.

TABLE OF CONTENTS

1 PROBLEM SOLVING

1.1 INTRODUCTION

Computer problem solving is an iteractive process requiring much thought, careful planning, logical precision, persistance and attention to detail. At the same time, it can be challenging, exciting and satisfying experience with personal creativity.

PROGRAMS AND ALGORITHM

A program is a set of explicit and unambiguous instructions expressed in a programming language. It takes the input from the end user and manipulate it according to its instructions and produces an output which represents the solution to the problem.

An algorithm consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.

1.2 PROBLEM SOLVING ASPECT

The problem solving is recognized as a creative process which largely defines systematization and mechanization.

The various aspects of problem solving are :

1. Problem definition phase
2. Getting started on a problem
3. The use of specific examples.
4. Simulation among problems.
5. Working backwards from the solution.
6. General problem - solving strategies.

1.2.1 Problem definition phase

The problem definition phase deals with "What must be done rather than how to do it". That is, the user tries to extract a set of precisely defined tasks from the problem statement.

1.2.2 Getting Started on a Problem

There are many ways to solve most problems and also many solutions to most problems, which makes the job of problem solving difficult to recognize quickly which paths are likely to be fruitless or productive. Hence the programmer do not have any idea where to start on the problem, even after the problem definition phase. They became concerned with details of the implementation before they have completely understood or worked out an implementation (i.e.,) independent solution. So it is better not to be concerned about detail in the beginning of the problem solving phase itself.

1.2.3. The use of specific examples

A useful stragegy is to use some heuristics to try to get a start with the problem. One way is to pick a specific example to the general problem that we wish to solve and try to workout the mechanism that will allow us to solve this particular problem. It is usually much easier to work out the details of a solution to a specific problem because the relationship between the mechanism and the particular problem is more clearly defined.

1.2.4 Similarities among problems

Another way to make a start on a problem is to bring as much past experience as possible to bear on the current problem. Therefore it is always make an effort to be aware of the similarities among problems. The more experience one has the more tools and techniques one can bring to bear in tackling a given problem. Sometimes, it leads to discovering desirable or better solution to a problem.

The important skill in problem solving is the ability to view a problem from a variety of angles. Once this skill has been established, then it should be possible to get started on any problem.

1.2.5 Working backwards from the solution

We can also start a problem by working backwards to the starting conditions. Documentation as we go along the various steps and explorations made allow us to systematize our investigation and avoid duplication of effort. The most critical thing of all in developing problem solving skills is its practice.

1.2.6 General problem solving strategies

There are a number of general and powerful computational strategies that are repeatedly used in various guises in computing science.

They are:

+ Divide and conquer
+ Dynamic programming
+ Greedy search
+ Back tracking
+ Branch and Bound Techniques.

The most widely known and most often used of these principles is the divide and conquer strategy, which divides the original problem into two or more sub-problems.

Another general problem solving strategy is Dynamic programming, which builds of intermediate steps. The techniques of greedy search, back tracking and branch-and-bound evaluations are all variations on the basic dynamic programming idea.

1.3 TOP-DOWN DESIGN

Top-down design or stepwise refinement is a strategy that can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program

implementation. It allows us to build our solutions to a problem in a stepwise fashion.

1.3.1 Breaking a problem into subproblems

The process of repeatedly breaking a tasks down into subtasks and then each subtask into still smaller subtasks and continue until we end up with subtasks that can be implemented as program statements. The larger and more complex the problem, the more it will need to be broken down to be made it tractable. This makes the implementation task as simple as possible.

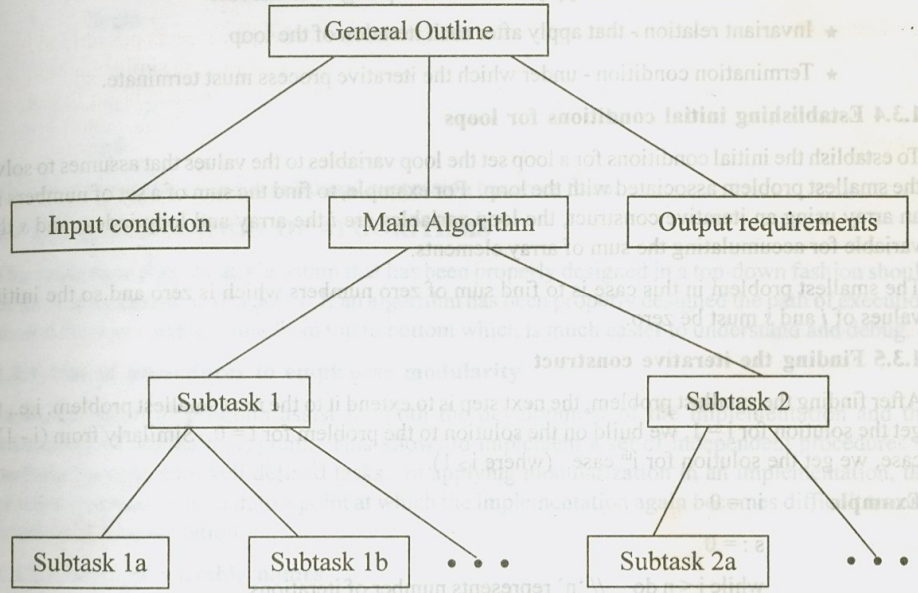


Fig. 1.3 Breakdown of a problem into substasks

1.3.2 Choice of suitable data structure

All programs operate on data and consequently the way the data is organised can have a effect on every aspect of the final solution.

An appropriate choice of data structure is usually leads to a simple, transparent and efficient implementation. Data structures and algorithms are usually linked to one another. A small change in data organisation can have a significant influence on the algorithm required to solve the problem.

Setting up data structures are based on the questions such as

1. Can it be easily searched ?
2. Can it be easily updated ?
3. Whether the data structure involve the excessive use of memory.
4. Can any one of the common data structures such as array, queue, set, stack, tree, graph,

list be used to formulate the problem ?

5. Will it reduce the amount of computation to find the intermediate results, which is based on the way in which it is arranged.

1.3.3 Construction of loops

To construct any loop, we must take three things in account. They are

- ★ Initial conditions - that apply before the loop begins to execute
- ★ Invariant relation - that apply after each iteration of the loop.
- ★ Termination condition - under which the iterative process must terminate.

1.3.4 Establishing initial conditions for loops

To establish the initial conditions for a loop set the loop variables to the values that assumes to solve the smallest problem associated with the loop. For example, to find the sum of a set of numbers in an array using an iterative construct, the loop variables are *i* the array and loop index, and *s* the variable for accumulating the sum of array elements.

The smallest problem in this case is to find sum of zero numbers which is zero and so the initial values of *i* and *s* must be zero.

1.3.5 Finding the iterative construct

After finding the smallest problem, the next step is to extend it to the next smallest problem. i.e., to get the solution for i = 1, we build on the solution to the problem for i = 0. Similarly from $(i - 1)^{th}$ case, we get the solution for i^{th} case. (where $i \geq 1$).

Example

```
i : = 0
s : = 0
while i < n do    // 'n' represents number of iterations
begin
        i : = i + 1;
        s : = s + a[i]
end
```

1.3.6 Termination of loops

The loops can be terminated in a number of ways. In general the termination conditions are dictated by the nature of the problem. The simplest condition for terminating a loop occurs when the number of iterations to be made is known in advance. For example, In Pascal the for-loop can be used as

```
for i : = 1 to n do
  begin
  .....
  .....
  end
```

This loop terminates unconditionally after n iterations.

The while-loop in Pascal can be used as

```
while (x > 10) do
  begin
  .....
  .....
  end
```

This loop terminates when conditional expression becomes false.

1.4 IMPLEMENTATION OF ALGORITHMS

The implementation of an algorithm that has been properly designed in a top-down fashion should be an almost mechanical process. If an algorithm has been properly designed the path of execution should flow in a straight line from top to bottom which is much easier to understand and debug.

1.4.1 Use of procedures to emphasise modularity

Modularization of program assist for both the development of the implementation and the readability of the main program. This allows to implement a set of independent procedures to perform specific and well defined tasks. In applying modularization in an implementation, the process is not taken too far, to a point at which the implementation again becomes difficult to read because of fragmentation.

1.4.2 Choice of variable names

Choosing appropriate variable and constant names makes programs more meaningful and easier to understand. Each variable should only have one role in a given program.

1.4.3 Documentation of programs

Another useful documenting practice that can be employed is to associate a brief but accurate comment with each procedure. A good programming practice is always to write programs so that they can be executed and used by other people unfamiliar about the program. This means that the program must specify during execution exactly what responses it requires from the user.

1.4.4 Debugging programs

- ◆ In implementing an algorithm it is very important to carry out a number of tests to ensure that the program is behaving correctly according to its specifications.
- ◆ To make the task of detecting logical errors, a set of statements that will print the information at strategic points in the computation is build into the program.
- ◆ The simplest way to implement this debugging tool is to have a Boolean variable (eg. debug) which is set to true when the debugging output for the program is required.

Each debugging output can then be parenthesized in the following way:

```
if debug then
  begin
    writeln (....)
    .....
    .....
  end
```

- A good rule to end follow when debugging is not to assume anything.

1.4.5 Program testing

In attempting to test whether or not a program will handle all variations of the problem it was designed to solve that it will cope with the limiting and unusual cases. We might check whether the program solves the smallest problem, whether it handles the case when all data values are the same and so on. Wherever possible, input and output assertions should be accompanied to the programs. We have to build the program that informatively respond to the user when it receives input conditions it was not designed to handle.

A good rule to follow is that fixed numeric constants should only be used in programs for things like the number of months in a year and so on.

1.5 PROGRAM VERIFICATION

The cost of software development is extremely high. Also, it may cause serious changes on sensitive data if the program doesn't work correctly. Therefore, it is essential to verify whether the program works correctly or not.

A better method of program verification is to verify the individual modules as it is completed rather than postponing the verification process to the end.(i.e.,) after integrating the modules.

The various phases in the program verifications are

1. Input and output assertion
2. Computer model for program execution
3. Implications and symbolic executions.
4. Verification of straight-line program segments
5. Verification of program segments with branches
6. Verification of program segments with loops
7. Verification of program segments that employ arrays
8. Proof of termination.

1.5.1 Input and Output assertions

The input and output assertion is the first and foremost step that has to be taken inorder to prove the correctness of a program. It provides a formal statement which consists of specifications interms

of the variables.

The formal statements gives two assertions

(a) An input assertion

(b) An output assertion

Input assertion

It specifies any constraints that are placed on the value of the input variables used by the program.

Example

In the division operation let us consider 'd' as the divisor. Hence it is clear that d cannot have the value 0. The input assertion is therefore $d \neq 0$.

Output assertion

It specifies the result symbolically that the program is expected to produce for the input data which satisfies the input assertion.

Example

In calculating the quotient q and the remainders x from the division of 'a' by 'b', then the output assertion is as $(a = q * b + r) \wedge (r < b)$

where '^' represents the logical AND operation.

1.5.2 Compute Model for program execution

A program may have a variety of execution paths leading to successful termination. For a given set of input conditions only one of these paths will be followed. The progress of a computation from specific input conditions through to termination can be thought of as a sequence of transitions from one computation state to another. Each state is defined by the values of all variables at the corresponding point in time.

A state transition model for program execution provides a foundation on which to construct correctness proof of algorithms.

1.5.3 Implications and Symbolic execution

Verifying a program can be formulated as a set of implications which must be shown to be logically true.

The general form of implication is

$P \supset Q$

P - Assumption

Q - Conclusion

P	Q	P ⊃ Q
True	True	True
True	False	False
False	True	True
False	False	True

Fig. 1.5(a) Truth Table defining implication

In order to show that these implications or propositions are logical true, it is necessary to use the technique of symbolic execution.

In symbolic execution, all input data values are replaced by symbolic values and all arithmetic operations on numbers are translated into algebraic manipulation of symbolic expression. This enables us to transform the verification procedure into providing that the input assertion with symbolic values substituted for all input variables implies the output assertion with final symbolic values substituted for all variables.

Step	Normal Execution	Symbolic Execution
	Input values : x = 3, y = 1	Input values : $x = \alpha$, $y = \beta$
1.	X : = x - y ⇒ x = 3 - 1 = 2	X : = x - y ⇒ $x = \alpha - \beta$
2.	Y : = x + y ⇒ y = 2 + 1 = 3	Y : = x + y ⇒ $y = (\alpha - \beta) + \beta = \alpha$
3.	X : = y - x ⇒ x = 3 - 2 = 1	X : = y - x ⇒ $x = (((\alpha - \beta) + \beta) - (\alpha - \beta)) = \beta$.

Fig. 1.5(6) Normal and Symbolic execution for exchange mechanism.

Consider the following program segment labelled with input and output assertions :

```
A    readln (x, y);
     {assert ; true)
     x : = x - y;
     y : = x + y;
     x : = y - x;
B    {assert  x = yo ∧ y = xo }
     where xo and yo refer to the initial values of x and y respectively.
```

Fig. 1.5(b) Exchange Mechanism program segment

1.5.4 Verification of Straight - line program segments

Let us consider the exchange mechanism program segment.

The verification condition is ;

VC (A - B) : true ⊃ { $x = yo \wedge y = xo$ } on substitution of initial and final values of all variables, we get :

$$VC\ (A - B) : \text{true} \supset ((\alpha - \beta) + \beta) - (\alpha - \beta))$$

$$= \beta \wedge ((\alpha - \beta) + \beta) = \infty$$

The conclusion part of the verification condition an be simplified to yield $\beta = \beta$ and $\alpha = \alpha$ which is true and so the implication is true.

[Note : VC (A - B) refers to the verification condition over the program segment A to B. Therefore apply the resultant of the program A to the program segment B.

(i.e) Symbolic execution of A is resulted as

$x = \alpha, y = \beta$ [initial values]

$x = ((\alpha - \beta) + \beta) - (\alpha - \beta)$ [Final Value]

$y = (\alpha - \beta) + \beta)$ [Final Value]

program segment B is { $x = yo \wedge y = xo$ }.

1.5.5 Verification of program segments with branches

To handle program segments that contain branches, it is necessary to set up and prove verification condition for each branch seperately. Let us consider the following program segment that ensures x is less than or equal to y.

```
     read ln (x, y)
A    {assert P_A : true}
     if x > y then
     begin
     t : = x;
     x : = y;
     y : = t;
     end.
B    {assert P_B : ((x < = y) ∧ (x = xo ∧ y = yo)) ∨ ( x = yo ∧ y = xo);
```

In general, the propositions that must be proved for the basic if construct are $P_A \wedge C_A \supset P_B$

$P_A \wedge \sim C_A \supset P_B$ where C_A is the branch condition.

1.5.6 Verification of Program segments with loops

Problems using program segments with loops can be solved with a loop invariant.

A Loop invariant is a property that captures the progressive computational role of the loop while at the same time remaining true, before and after each loop traversal irrespective of how many times the loop is executed. Let us consider the following single - loop program structure as model.

A {input assertion P_A}

```
--
--
Straight line program segment
--
--
```

B {loop invariant I_B}

while loop - condition do

begin

```
--
--
Loop-free program segment
--
--
```

end

C {Output assertion P_C}

The steps to be employed to verify a loop segment are.

Step 1: We have to show that the loop invariant is initially true. This can be done by setting up a verification condition,

VC (A - B) for the program segment from A to B.

Step 2 : To show that the loop invariant is still true after the segment of program within the loop has been executed. To do this, set up a verification condition as VC (B - B).

The loop invariant with initial values of variables set, together with condition for loop execution C_ρ, implies the truth of the loop invariant with final values of variables.

(i.e) $I_B \wedge C_B \supset I_B$.

Step 3 : The final step is verifying a loop segment, it is necessary to show that, the loop invariant together with the negation of the loop entry condition, implies the assertion that applies on existing form of the loop. The verification condition for this care is VC (B - C). The corresponding proposition will be :

$$I_B \wedge \sim C_B \supset P_C$$

1.5.7 Verfication of program segments that employ arrays

The idea of symbolic execution can be extended to simpler examples that employ arrays by accounting symbolic values of all array elements.

Let us consider the program segment which finds the position of the smallest element in the array.

A {assert $P_A : n \geq 1$}

i : = 1;

p : = 1;

B $\{in\,variant\ I_B : (1 \leq i \leq n) \wedge (1 \leq p \leq i) \wedge (a[p] \leq a[1], a[2], ..., a[i])\}$

while i < n do

begin

i : = i + 1;

if a[i] < a [p] then p! = i

end

C $\{assert\ P_C : (1 \leq p \leq n) \wedge n(a[p] \leq a[1], a[2], ..., a[n])\}$

Assume the initial values of a[1], a[2], ..., a[n] as $\alpha_1, \alpha_2, ...\alpha_n$ and the initial value of n as δ, then the symbolic execution to check the verification condition is given as :

$$VC(A\text{-}B) : \delta \geq 1 \supset (1 \leq 1 \leq \delta) \wedge (1 \leq 1 \leq 1) \wedge \alpha_1 \leq \alpha_1$$

1.5.8 Proof of termination

To prove that a program terminates, it is necessary to show that every loop in the program terminates in a finite number of steps. Consider, for example, the for - loop ;

```
for i = 1 to n do
begin
...
...
end
```

when n is positive and finite, the loop is guaranteed to terminate because, with each iteration, the number of steps to the termination condition being satisfied is reduced by at least one. This reduction can only be carried out a finite number of times and so the loop must terminate.

The proof of termination is much more subtle and elusive for loops which has any one of the following two situations.

* When there is no single variable that is monotonically progressing towards the termination condition.
* An arithmetic combination of two or more variables makes progress towards termination with each iteration.

The problem of proving loop termination can often be approached by associating another expression in addition to the loop invariant, with each loop.

The expression $\in$, should be chosen to be a function of the variables used in the loop. It should always be non - negative and the value is decreased with each loop iteration.

To verify the loop structure perform the following steps.

Step 1 : Show that the loop invariant I_B, together with condition for loop execution C_B, implies that the value of the expresion $\in$ is greater than zero.

(i.e)

$$TC1(B): I_B \wedge C_B \supset \in > 0$$

The condition $\in \geq 0$ becomes an invariant of the loop.

Step 2 : Show that the loop invariant I_B, together with the condition for loop execution, C_B, implies that the value $\in_0$ of the expression before execution is strictly greater than its value $\in$ after loop execution. (i.e) for a loop B $TC2(B-B): I_B \wedge C_B \supset (\in_0 > \in) \wedge (\in \geq 0)$

1.6 THE EFFICIENCY OF ALGORITHMS

The Efficiency considerations for algorithms are inherently tied in with the design, implemenation and analysis of algorithms. The resources used by the algorithms are central processing unit (CPU) time and internal memory. So the efficiency of the algorithm lies with the economical usage of these resources.

Some of the suggestions to improve the efficiency of an algorithm in designing are

* Redundant computations
* Referencing array elements
* Inefficiency due to late termination
* Early detection of desired output conditions.
* Trading storage for efficiency gains.

1.6.1 REDUNDANT COMPUTATONS

The effects of redundant computations are most serious when they are embedded within a loop that must be executed many times which utilizes unnecessary memory space. The most common mistake using loops is to repeatedly recalculate part of an expression, that remains constant throughout the entire execution phase of the loop. For example

```
S : = 0; K : = 2;
for i = 1 to n do
begin
        S : = k * k * k + i ;
end
```

The unnecessary multiplication can be removed by precomputing a constant K_3 before executing the loop.

```
S : = 0 ; K : = 2 ;
K3 : = k * k * k ;
for i = 1 to n do
begin
        S : = K3 + i ;
end
```

1.6.2 Referencing array elements

Let us consider for example, the two versions of an algorithm for finding the maximum.

Version (1)

```
for i = 2 to n do
    if a [i] > a [p] then
        max : = a [p]
```

version (2)

```
max : = a[1];
for i = 2 to n do
    if a[i] > max then
        max : = a[i];
```

The version (2) implementation would normally preferred because the condition test a[i] > max is more efficient because it uses the variable max which requires only one memory reference instruction, whereas the use of variable a[p] requires two memory references.

1.6.3 Inefficiency due to late termination

Inefficiencies can come into an implementation due to late termination.

Let us consider an example to search an alphabetically ordered list of names for some particular name linearly.

Suppose we were looking for the name 'Ram', then as soon as we had encountered a name that occurred alphabetically later than Ram, (ie) Rekha, we would no need to proceed further.

The inefficient implementation could have the form :

```
while (namesought < > current name and not end of file) do
        get next name from list
```

A more efficient implementation would be :

```
while (namesought > current name and not end of file) do
        get nextname from list.
```

1.6.4 Early detection of desired output conditions

Due to the nature of the input data, the algorithm establishes the desired output condition before the general conditions for termination have been met.

For example, a bubble sort might be used to sort a set of data that is already almost in sorted order. If there have been no exchanges in the current pass, the data must be sorted and so early termination can be applied.

1.6.5 Trading storage for efficiency gains

Storage and efficiency is often used to improve the performance of an algorithm.

One strategy that sometimes used to speed up an algorithm is to implement the algorithm using the least number of loops, which makes the program much harder to read and debug. It is therefore usually better to have one loop for one job.

1.7 THE ANALYSIS OF ALGORITHMS

The analysis of algorithms is made considering both qualitative and quantitative aspects to get a solution that is economical in the use of computing and human resources which improves the performance of an algorithm. A good algorithm usually possess the following qualities and capabilities.

1. They are simple but powerful and general solutions.
2. They are user friendly
3. They can be easily updated.
4. They are correct
5. They are able to be understood on a number of levels.
6. They are economical in the use of computer time, storage and peripherals.
7. They are well documented.
8. They are independent to run on a particular computer.
9. They can be used as subprocedures for other problems.
10. The solution is pleasing and satisfying to its designer.

1.7.1 Computational Complexity

The computational complexity can be measured in terms of space and time required by an algorithm.

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run the algorithm.

Time Complexity

The time complexity of an algorithm is the amount of time it needs to run the algorithm.

The time taken by the program is the sum of the compile time and run time.

To make a quantitative measure of an algorithm's performance, the computational model must capture the essence of the computation while at the same time it must be divorced from any programming language.

1.7.2 Asymtotic Notation

Asymtotic notations are method used to estimate and represent the efficiency of an algorithm using simple formula. This can be useful for seperating algorithms that leads to different amounts of work for large inputs.

Comparing or classify functions that ignores constant factors and small inputs is called as asymtotic growth rate or asymtotic order or simply the order of functions. Complexity of an algorithm is usually represented in O, o, θ, Ω notations.

Big - oh notation (O)

This is a standard notation that has been developed to represent functions which bound the computing time for algorithms and it is used to define the worst case running time of an algorithm and concerned with very large values of N.

Definition : - $T(N) = O(f(N))$, if there are positive constants C and n_o such that $T(N) \leq Cf(N)$ when $N \geq n_o$

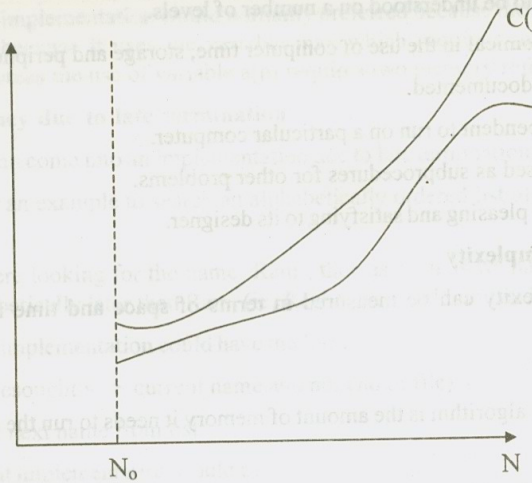


Fig. 1.7(a) Big - oh notation T(N) ∈ O(F(N))

BIG - OMEGA NOTATION (Ω)

This notation is used to describe the best case running time of algorithms and concerned with very large values of N.

Definition : - T(N) = Ω(f(N)), if there are positive constants C and n_o such that T(N) ≥ CF(N) when $N \geq n_o$

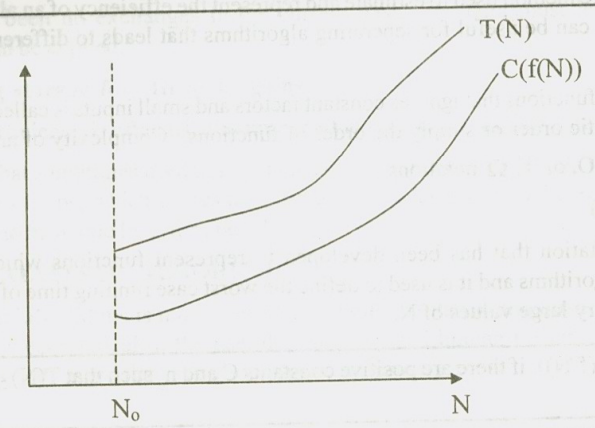


Fig. 1.7(b) BIG - OMEGA NOTATION $T(N) \in \Omega(F(N))$

BIG - THETA NOTATION (θ)

This notation is used to describe the average case running time of algorithms and concerned with very large values of n.

Definition : - T(N) = θ(F(N)), if there are positive constants C_1, C_2 and n_o such that T(N) = O(F(N)) and T(N) = Ω (F(N)) for all $N \geq n_o$.

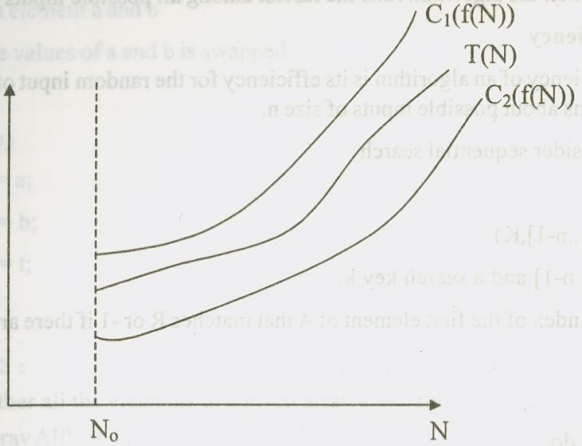


Fig. 1.7 (c) BIG - THETA NOTATION $T(N) \in \theta(F(N))$

Little - Oh Notation (o)

This notation is used to describe the worstcase analysis of algorithms and concerned with small values of n.

Definition : $T(N) = o(F(N))$ *if* $T(N) = O(F(N))$ *and* $T(N) \neq \theta(T(N))$

Basic Asymptotic Efficiency Classes

Computing Time	Name
0(1)	constant
0(log n)	Logarithmic function
0(n)	Linear
$0(n^2)$	quadratic
$0(n^3)$	cubic
$0(2^n)$	exponential
0(nlogn)	n - log - n Logarithmic
n!	factorial

1.7.3 WORST - CASE, BEST - CASE AND AVERAGE - CASE EFFICIENCIES

Worst - Case - Efficiency

The worst - case efficiency of an algorithm is its efficiency for the worst - case input of size n, which is an input of size n for which the algorithm runs the longest among all possible inputs of that size.

Best - Case Efficiency

The best - case efficiency of an algorithm is its efficiency for the best case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.

Average - Case Efficiency

The average - case efficiency of an algorithm is its efficiency for the random input of size n, which makes some assumptions about possible inputs of size n.

For example, let us consider sequential search

ALGORITHM

SequentialSearch (A[0...n-1],K)

// Input : An array A[0..n-1] and a search key k.

// Output : Returns the index of the first element of A that matches R or -1 if there are no matching elements.

$i \rightarrow 0$

while i < n and A[i] # k do

i $\rightarrow$ i + 1

if i < n return i

else return - 1

Here, the best - case efficiency is 0(1) where the first element is itself the search element and the worst - case efficiency is 0(n) where the last element is the search element or the search element may not be found in the given array.

1.8 SAMPLE ALGORITHMS

PROBLEM 1 (a) :

Give two variables a and b, exchange the values assigned to them without using temporary variable.

INPUT : Given element a and b

OUTPUT : The value of b is stored in variable a and the value of a is stored in variable b.

ALGORITHM :

```
Swap (a, b)
    a = a + b;
    b = a - b;
    a = a - b;
    return a, b
```

PROBLEM 1 (b)

Give two variables a and b, exchange the values assigned to them using temporary variable.

INPUT : Given element a and b

OUTPUT : The values of a and b is swapped

ALGORITHM:

```
Swap (a, b)
    t = a;
    a = b;
    b = t;
return a, b
```

PROBLEM 2 :

To check whether all the elements in a given array are distinct.

INPUT : An array A[0...n-1]

OUTPUT : Returns "true" if all the elements in A are distinct and "false" otherwise.

ALGORITHM:

```
UniqueElements (A[0...n-1])
    for i = 0 to n-1 do
        for j = i + 1 to n-1 do
            if A[i] = = A[j]
            return false
    return true
```

PROBLEM 3 :

Make a count of the number of students on a particular subject who passed the examination scoring 50 and above.

INPUT : Number of Students (n)

OUTPUT : Total count of the students who passed in that subject.

ALGORITHM:

```
Passcount (n)
    count = 0;
    i = 0;
    while (i < n) do
        read (m)
```

```
            if m > = 50 then
                    count = count + 1;
                i = i + 1;
                endwhile
        return count
```

PROBLEM 4(a) :

To find the sum of a given set of numbers.

INPUT : An array A[0.. n - 1]

OUTPUT : returns sum of the elements in the given array.

ALGORITHM:

```
Sum1(A [0 .. n-1]
    sum = 0;
        for i = 0 to n -1 do
            sum = sum + A[i]
return sum
```

PROBLEM 4(b) :

To find the sum of the squares of a given set of numbers.

INPUT : An array A[0.. n - 1]

OUTPUT : Returns the sum of squares of a given set of numbers.

ALGORITHM:

```
Sumsquare (A[0.. n - 1])
    sum = 0;
        for i = 0 to n - 1 do
            sum = sum + A[i] * A[i];
return sum
```

PROBLEM 5(a) :

To compute factorial of a given number without recursion.

INPUT : The element n

OUTPUT : Returns factorial of the given element n.

ALGORITHM:

```
fact (n)
    f = 1;
    for (i = 1; i < = n; i ++)
        f = f * i;
    return f
```

PROBLEM 5(b) :

To compute factorial of a given number using recursion.

INPUT : A positive integer n

OUTPUT : Returns the factorial of the given element.

ALGORITHM:

```
fact (n)
    if (n = = 0)
        return 1
    else
        return fact (n - 1) * n
```

PROBLEM 6(a) :

Generate the Fibonacci series for first n terms.

INPUT : A positive integer n

OUTPUT : Prints Fibonacci series for n terms.

ALGORITHM:

```
Fibo (n)
    F1 = -1;    F2 = 1;
    Fib = 0;
        for i = 1 to n do
            Fib = F1 + F2 ;
            write Fib
            F1 = F2;
            F2 = Fib;
```

PROBLEM 6(b) :

Generate n^{th} Fibonacci term using recursion.

INPUT : A positive integer n

OUTPUT : The n^{th} Fibonacci number.

ALGORITHM:

```
Fibo (n)
    if n ≤ 1
        return n
    else
        return Fibo(n-1) + Fibo(n-2)
```

PROBLEM 7(a) :-

Design an algorithm to evaluate the function sin(x) as defined by the infinite series expansion.

$$Sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

INPUT : - Get the values for x and n

OUTPUT : - Sine function for the given terms is calculated.

ALGORITHM:-

```
Sineseries (x, n) // Get x in radians
S = 0;
term = x;
i = 1;
while i <= n do
    S = S + term;
    term = (term * x * x * (-1)) / ((i + 1) * (i + 2));
    i = i + 2
write S;
```

PROBLEM 7(b) :-

The exponential growth constant e is characterized by the expression

$$e = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...$$

Device an algorithm to compute e to n terms.

INPUT : - Get the values of n (no. of terms)

OUTPUT : - Sum of the exponential series for 'n' terms is calculated.

ALGORITHM:-

```
EXPOSERIES (n)
i = 1;
e = 1;
f = 1;
while i <= n do
    e = e + 1/f ;
    f = f * i ;
    i = i + 1;
write e ;
```

PROBLEM 8 :-

To sort the given set of numbers.

INPUT : - An array A[0..n - 1]of orderable element.

OUTPUT : - Array A[0..n - 1] sorted in ascending order.

ALGORITHM:-

```
Sort (A[0.. n - 1]
    for i = 0 to n - 2 do
        for j = 0 to n - 2 - i do
            if A[j + 1] < A[j]
                swap A[j] and A[j + 1]
```

PROBLEM 9 :-

To search a given element in the array using binary search non recursively.

INPUT : - An array A[0..n - 1]sorted in ascending order and a search key k.

OUTPUT : - An index of the array's element that is equal to K or -1 if there is no such element.

ALGORITHM:-

```
Binary Search (A[0 .. n - 1], K)
    l = 0;
    r = n - 1;
    while l ≤ r do
        m = (l + r) / 2
        if K = A[M]
```

```
                return m
            else if K < A[M]
                r = m - 1
            else
                l = m + 1
        return - 1
```

PROBLEM 10 :-

Design an algorithm that accepts a positive integer and reverse the order of its digits.

INPUT : - A positive integer n.

OUTPUT : - A positive integer n with reversed order of its digits.

ALGORITHM :-

```
Reverse (n)
    s = 0;
    while n > 0 do
        r = n % 10 ;
        s = r + s * 10;
        n = n/10;
    write s
```

PROBLEM 11 :-

To compute the multiplication of two square matrices of order n.

INPUT : - Two n x n matrices A and B.

OUTPUT : - Matrix C = AB

ALGORITHM :-

```
MatrixMult (A[0..n - 1, 0.. n - 1], B[0..n - 1, 0..n -1])
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            C[i, j] = 0;
            for k = 0 to n - 1 do
                C[i, j] = C[i, j] + A[i, k] * B[k, j];
    return C
```

PROBLEM 12(a) :-

To convert the given decimal number into its binary form.

INPUT : - A positive integer n.

OUTPUT : - A binary representation for the given decimal number n.

ALGORITHM :-

```
Decitobin (n)
    C = 0;
    while n > 0 do
        r = n% 2;
        a [i] = r;
        n = n/2 ;
        C = C + 1;
    endwhile
        for i = C - 1 to 0
        write a[i];
```

PROBLEM 12(b) :-

To convert a binary number into a decimal number.

INPUT : - A positive integer in binary form.

OUTPUT : - decimal form of the given number is obtained.

ALGORITHM :-

```
Bintodeci (n)
    q = n;
    s = 0;
    i = 0;
    while q > 0 do
        r = q % 10 ;
        s = s + r * pow (2, i);
        i = i + 1;
    endwhile
    write s ;
```

Questions

Part - A

1. Define an algorithm
2. Write an algorithm to swap two numbers.
3. Define the top - down design strategy.
4. What is meant by problem solving?
5. Mention some of the problem solving strategies.
6. What is divide and conquer strategy?
7. What are the steps involved in problem solving?
8. Write an algorithm to find the factorial of a given number.
9. List the types of control structures.
10. Define the worst case and average case complexities of an algorithm.
11. How do you debug a program?
12. What is program testing?
13. What is program verification?
14. Write down the various phases in the program verification.
15. Define space complexity and time complexity.
16. What are the qualities and capabilities of good algorithm.
17. How can you improve an efficiency of an algorithm in the design phase?
18. What do you mean by proof of termination?
19. What is O - notation?
20. Write an algorithm to find the sum of a set of numbers.

Part - B

1. Explain Top - down design strategy in detail.
2. Write down the algorithm for binary search.
3. The exponential growth constant e is characterized by the expression

   $e = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots\ldots$. Device an algorithm to compute e to n terms.
4. Explain in detail the types of analysis that can be performed on an algorithm.
5. Write an algorithm to perform matrix multiplication and give the analysis for it.
6. Design an algorithm to compute the sum of the squares of n numbers.
7. Design an algorithm that accepts a positive integer and reverses the order of digits.
8. Write an algorithm to convert a decimal number to its corresponding octal form.

ABSTRACT DATA TYPE

In programming each program is breakdown into modules, so that no routine should ever exceed a page. Each module is a logical unit and does a specific job modules which inturn will call another module.

Modularity has several advantages

1. Modules can be compiled separately which makes debugging process easier.
2. Several modules can be implemented and executed simultaneously.
3. Modules can be easily enhanced.

Abstract Data type is an extension of modular design.

An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,

The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

2.1 THE LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, \ldots\ldots, A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) - Returns the position of its predecessor i-1.
6. Print list - Contents of the list is displayed.
7. Makeempty - Makes the list empty.

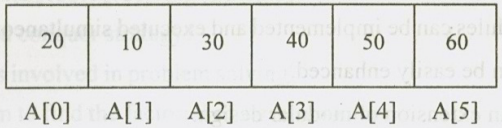
2.1 .1 Implementation of List ADT

1. Array Implementation
2. Linked List Implementation
3. Cursor Implementation.

Array Implementation of List

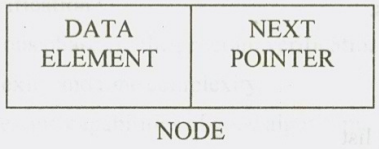
Array is a collection of specific number of data stored in a consecutive memory locations.

* Insertion and Deletion operation are expensive as it requires more data movement
* Find and Printlist operations takes constant time.
* Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.



Linked List Implementation

Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.



NODE

Insertion and deletion operations are easily performed using linked list.

Types of Linked List

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List.

2.1.2 Singly Linked List

A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

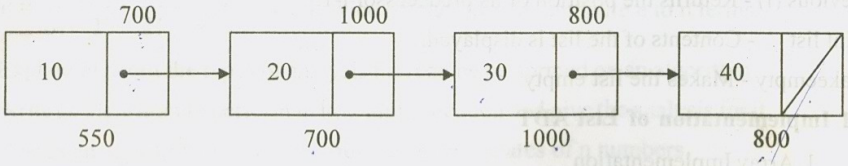


Fig. 2.1 LINKED LIST

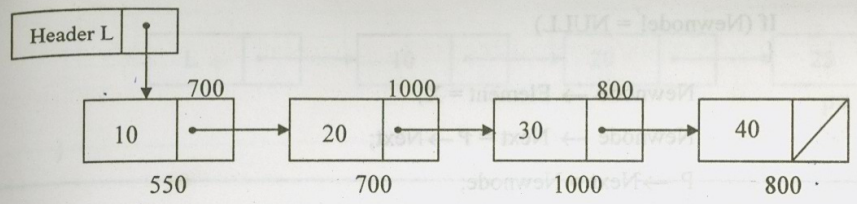


Fig. 2.1 LINKED LIST WITH A HEADER

DECLARATION FOR LINKED LIST

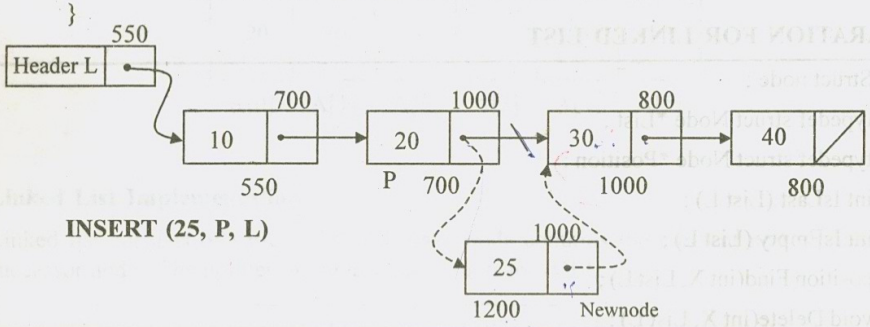
```
Struct node ;
typedef struct Node *List ;
typedef struct Node *Position ;
int IsLast (List L) ;
int IsEmpty (List L) ;
position Find(int X, List L) ;
void Delete(int X, List L) ;
position FindPrevious(int X, List L) ;
position FindNext(int X, List L) ;
void Insert(int X, List L, Position P) ;
void DeleteList(List L) ;
Struct Node
{
        int element ;
        position Next ;
};
```

ROUTINE TO INSERT AN ELEMENT IN THE LIST

```
void Insert (int X, List L, Position P)
        /* Insert after the position P*/
{

position Newnode;
Newnode = malloc (size of (Struct Node));
```

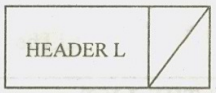
```
        If (Newnode! = NULL)
        {
                Newnode → Element = X;
                Newnode → Next = P→Next;
                P →Next = Newnode;
        }
}
```



INSERT (25, P, L)

ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

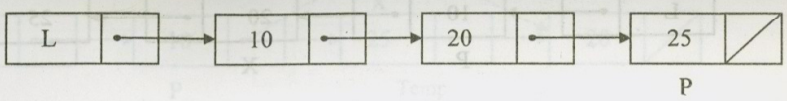
```
int IsEmpty (List L) /*Returns 1 if L is empty */
{
        if (L →Next = = NULL)
          return (1);
}
```



Empty List

ROUTINE TO CHECK WHETHER THE CURRENT POSITION IS LAST

```
int IsLast (position P, List L) /* Returns 1 is P is the last position in L */
{
        if (P→Next = = NULL)
                return (1);
```

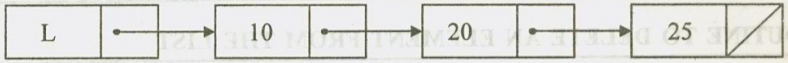



```
}
```

FIND ROUTINE

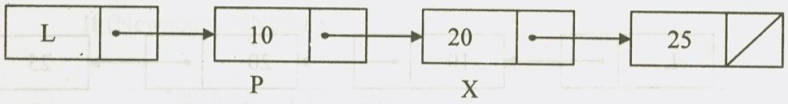
```
position Find (int X, List L)
{
        /*Returns the position of X in L; NULL if X is not found */
        position P;
        P = L → Next;
        while (P! = NULL && P→Element ! = X)
                P = P→Next;
        return P;
}
```

FIND (10)



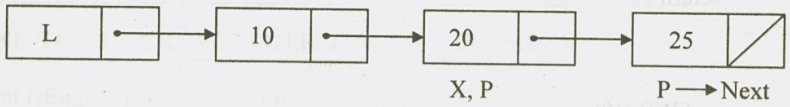
FIND PREVIOUS ROUTINE

```
position FindPrevious (int X, List L)
{
        /* Returns the position of the predecessor */
        position P;
        P = L;
        while (P → Next ! = Null && P →Next →Element ! = X)
                P = P → Next;
        return P;
}
```



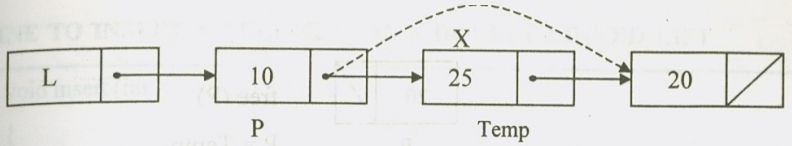
FINDNEXT ROUTINE

```
position FindNext (int X, List L)
{
        /*Returns the position of its successor */
        P = L → Next;
        while (P → Next! = NULL  && P → Element ! = X)
                P = P → Next;
        return P → Next;
}
```

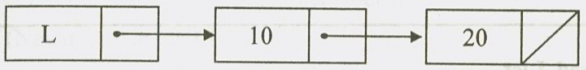


ROUTINE TO DELETE AN ELEMENT FROM THE LIST

```
void Delete(int X, List L)
{
        /* Delete the first occurence of X from the List */
        position P, Temp;
        P = Findprevious (X,L);
        If (!IsLast(P,L))
        {
                Temp = P → Next;
                P → Next = Temp → Next;
                Free (Temp);
        }
}
```



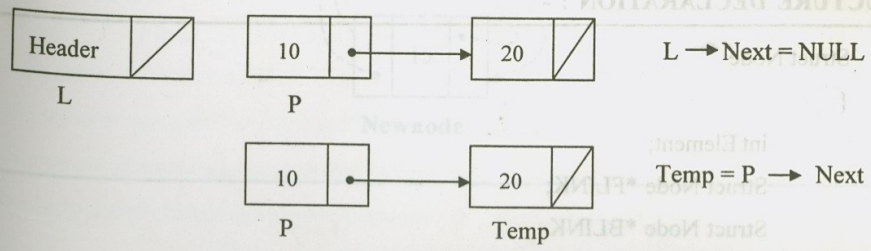
BEFORE DELETION

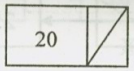


AFTER DELETION

ROUTINE TO DELETE THE LIST

```
void DeleteList (List L)
{
        position P, Temp;
        P = L → Next;
        L → Next = NULL;
        while (P! = NULL)
        {
                Temp = P → Next
                free (P);
                P = Temp;
        }
}
```





free (P)

P = Temp

Temp, P = NULL

2.1.3 Doubly Linked List

A Doubly linked list is a linked list in which each node has three fields namely data field, forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node.

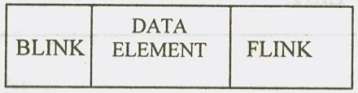


Fig. 2.1.3 (a) NODE IN DOUBLY LINKED LIST

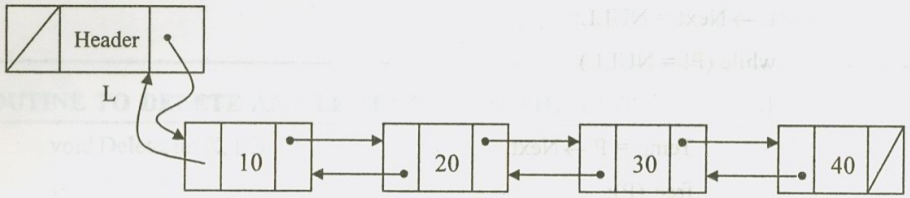


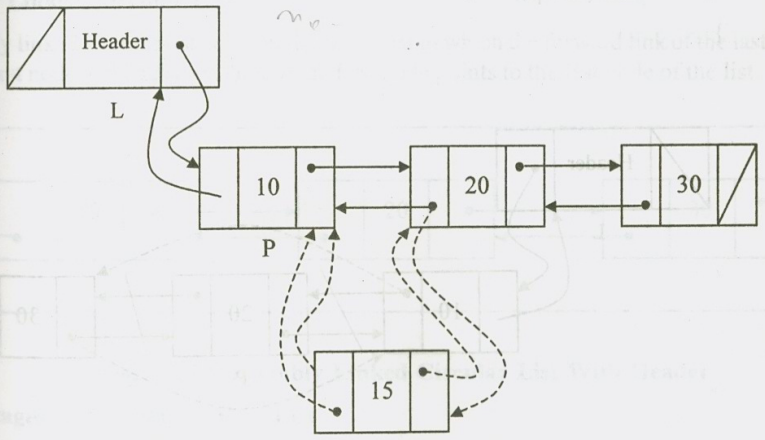
Fig. 2.1.3 (b) DOUBLY LINKED LIST

STRUCTURE DECLARATION : -

```
Struct Node
{
        int Element;
        Struct Node *FLINK;
        Struct Node *BLINK
};
```

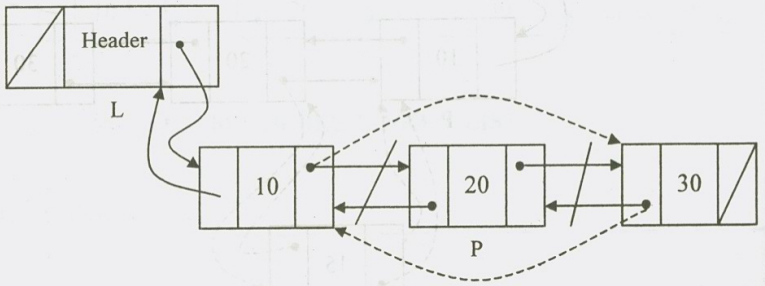
ROUTINE TO INSERT AN ELEMENT IN A DOUBLY LINKED LIST

```
void Insert (int X, list L, position P)
{
        Struct  Node * Newnode;
        Newnode = malloc (size of (Struct Node));
        If (Newnode ! = NULL)
        {
                Newnode → Element = X;
                Newnode → Flink = P → Flink;
                P → Flink → Blink = Newnode;
                P → Flink = Newnode ;
                Newnode → Blink = P;
        }
}
```



ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
        position P;
        P = Find (X, L);
        If ( IsLast (P, L))
        {
                Temp = P;
                P → Blink →Flink = NULL;
                free (Temp);
        }
        else
        {
                Temp = P;
                P → Blink →Flink = P →Flink;
                P →Flink →Blink = P →Blink;
                free (Temp);
        }
}
```



Advantage

* Deletion operation is easier.
* Finding the predecessor & Successor of a node is easier.

Disadvantage

* More Memory Space is required since it has two pointers.

2.1.4 Circular Linked List

In circular linked list the pointer of the last node points to the first node. Circular linked list can be implemented as Singly linked list and Doubly linked list with or without headers.

Singly Linked Circular List

A singly linked circular list is a linked list in which the last node of the list points to the first node.

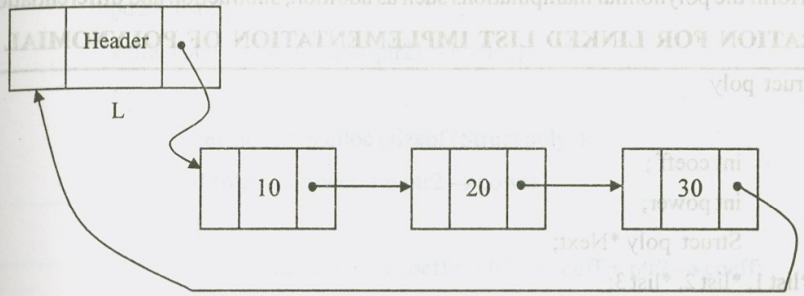


Fig. 2.1.4 Singly Linked Circular List With Header

Doubly Linked Circular List

A doubly linked circular list is a Doubly linked list in which the forward link of the last node points to the first node and backward link of the first node points to the last node of the list.

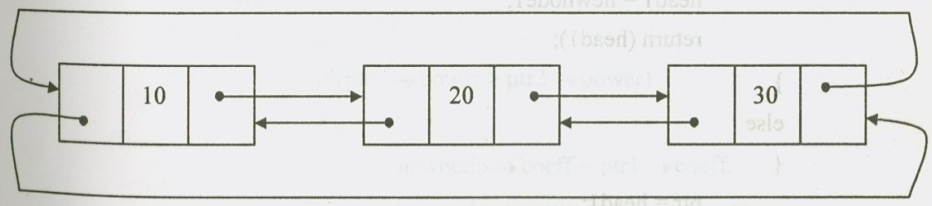


Fig. 2.1.4 (b) Doubly Linked Circular List With Header

Advantages of Circular Linked List

- It allows to traverse the list starting at any point.
- It allows quick access to the first and last records.
- Circularly doubly linked list allows to traverse the list in either direction.

2.1.5 Applications of Linked List

1. Polynomial ADT
2. Radix Sort
3. Multilist

Polynomial ADT

We can perform the polynomial manipulations such as addition, subtraction and differentiation etc.

DECLARATION FOR LINKED LIST IMPLEMENTATION OF POLYNOMIAL ADT

```
Struct poly
{
        int coeff ;
        int power;
        Struct poly *Next;
}*list 1, *list 2, *list 3;
```

CREATION OF THE POLYNOMIAL

```
poly create (poly *head1, poly *newnode1)
{
        poly *ptr;
        if (head1= =NULL)
        {
                head1 = newnode1;
                return (head1);
        }
        else
        {
                ptr = head1;
                while (ptr → next ! = NULL)
                ptr = ptr → next;
                ptr → next = newnode1;
        }
        return (head1);
}
```


ADDITION OF TWO POLYNOMIALS

```
void add ( )
{
        poly *ptr1, *ptr2, *newnode;
        ptr1 = list1;
        ptr2 = list2;
        while (ptr1! = NULL && ptr2! = NULL)
        {
                newnode = malloc (sizeof (Struct poly));
                if (ptr1 → power = = ptr2 → power)
                {
                        newnode → coeff = ptr1 → coeff + ptr2 → coeff;
                        newnode → power = ptr1 → power;
                        newnode → next = NULL;
                        list 3 = create (list3, newnode);
                        ptr1 = ptr1 → next;
                        ptr2 = ptr2 → next;
                }
                else
                {
                        if (ptr1 → power > ptr2 → power)
                        {
                                newnode → coeff = ptr1 → coeff;
                                newnode → power = ptr1 → power;
                                newnode → next = NULL;
                                list3 = create (list3, newnode);
                                ptr1 = ptr1 → next;
                        }
```

```
            else
            {
                    newnode → coeff = ptr2 → coeff;
                    newnode → power = ptr2 → power;
                    newnode → next = NULL;
                    list3 = create (list3, newnode);
                    ptr2 = ptr2 → next;
            }
    }
}
```

SUBTRACTION OF TWO POLYNOMIAL

```
void sub ( )
{
    poly *ptr1, *ptr2, *newnode;
    ptr1 = list1 ;
    ptr2 = list 2;
    while (ptr1! = NULL && ptr2! = NULL)
    {
        newnode = malloc (sizeof (Struct poly));
        if (ptr1 → power = = ptr2 → power)
        {
            newnode → coeff = (ptr1 → coeff) - (ptr2 → coeff);
            newnode → power = ptr1 → power;
            newnode → next = NULL;
            list3 = create (list 3, newnode);
            ptr1 = ptr1 → next;
            ptr2 = ptr2 → next;
        }
        else
        {
            if (ptr1 → power > ptr2 → power)
            {
                newnode → coeff = ptr1 → coeff;
                newnode → power = ptr1 → power;
                newnode → next = NULL;
                list 3 = create (list 3, newnode);
                ptr1 = ptr1 → next;
            }
            else
            {
                newnode → coeff = - (ptr2 → coeff);
                newnode → power = ptr2 → power;
                newnode → next = NULL;
                list 3 = create (list 3, newnode);
                ptr2 = ptr2 → next;
            }
        }
    }
}
```

POLYNOMIAL DIFFERENTIATION

```
void diff ( )
{
    poly *ptr1, *newnode;
    ptr1 = list 1;
    while (ptr1 ! = NULL)
    {
        newnode = malloc (sizeof (Struct poly));
        newnode → coeff = ptr1 → coeff *ptr1 → power;
        newnode → power = ptr1 → power - 1;
        newnode → next = NULL;
        list 3 = create (list 3, newnode);
        ptr1 = ptr1 → next;
    }
}
```

Radix Sort : - (Or) Card Sort

Radix Sort is the generalised form of Bucket sort. It can be performed using buckets from 0 to 9.

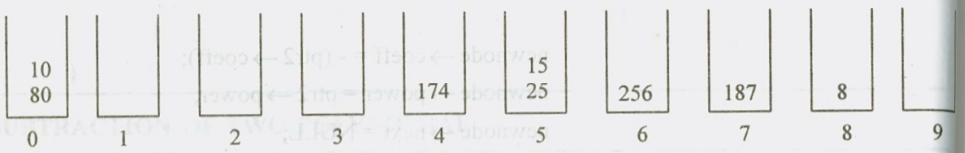
In First Pass, all the elements are sorted according to the least significant bit.

In second pass, the numbers are arranged according to the next least significant bit and so on this process is repeated until it reaches the most significant bits of all numbers.

The numbers of passes in a Radix Sort depends upon the number of digits in the numbers given.

PASS 1 :

INPUT : 25, 256, 80, 10, 8, 15, 174, 187

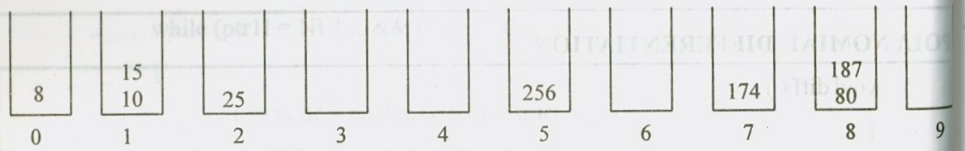


Buckets

After Pass 1 : 80, 10, 174, 25, 15, 256, 187, 8

PASS 2 :

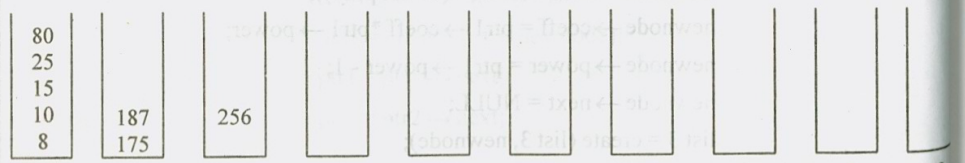
INPUT : 80, 10, 174, 25, 15, 256, 187, 8



After Pass 2 : 8, 10, 15, 25, 256, 174, 80, 187

PASS 3 :

INPUT : 8, 10, 15, 25, 256, 174, 80, 187



After pass 3 : 8, 10, 15, 25, 80, 175, 187, 256

Maximum number of digits in the given list is 3. Therefore the number of passes required to sort the list of elements is 3.

Multi Lists

More complicated application of linked list is multilist. It is useful to maintain student registration, Employee involved in different projects etc., Multilist saves space but takes more time to implement.

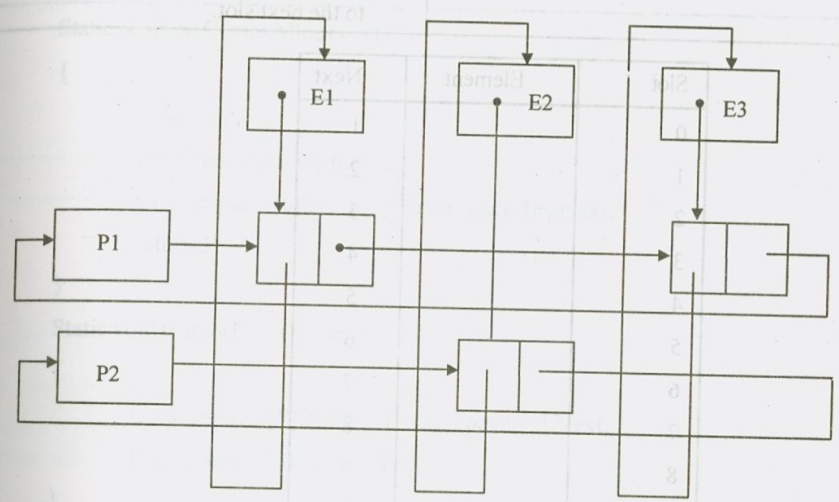


Fig. 2.1.5 Multilist Implementation For Employee Project Allocation

An employee can involve in any number of projects and each project can be implemented by any number of employees.

Employee E_1 is working on project P_1, E_2 is working on project P_2 & E_3 is working on project P_1. Project P_1 is implemented by the Employees E_1 & E_3. Project P_2 is implemented by the Employee E_2.

2.1.6 Cursor Implementation of Linked Lists

Cursor implementation is very useful where linked list concept has to be implemented without using pointers.

Comparison on Pointer Implementation and Curson Implementation of Linked List.

Pointer Implementation	Cursor Implementation
1. Data are stored in a collection of structures. Each structure contains a data and next pointer.	Data are stored in a global array of structures. Here array index is considered as an address.

Pointer Implementation	Cursor Implementation
2. Malloc function is used to create a node and free function is used to released the cell.	It maintains a list of free cells called cursors space in which slot 0 is considered as a header and Next is equivalent to the pointer which points to the next slot.

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		0

Fig. 2.1.6 Initialized Cursorspace

Declaration

```
typedef int ptrtoNode;
typedef ptrtoNode position;
void Initialize cursorspace (void);
int IsEmpty (List L);
int IsLast (position P, List L);
position Find (int X, List L);
void Delete (int X, List L);
void Insert (int X, List L, position P);
Struct Node
{
        int Element;
        position Next;
};
Struct Node Cursorspace [space size];
```

ROUTINE FOR CURSOR ALLOC & CURSOR FREE

```
Static position CursorAlloc (void)
{
        position P;
        P = CursorSpace [0].Next;
        CursorSpace [0].Next = CursorSpace [P].Next;
        return P;
}
Static void CursorFree (position P)
{
        CursorSpace [P].Next = CursorSpace [0].Next;
        CursorSpace [0].Next = P;
}
```

ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

```
int IsEmpty (List L)
{       /* Returns 1 if the list is Empty */.
        if (Cursorspace [0].  Next = = 0)
        return (1);
}
```

ROUTINE FOR ISLAST

```
int IsLast (Position P, List L)
{       /* Returns 1 if the p is in last position */
        if (CursorSpace [P].Next = = 0)
        return (1);
}
```

ROUTINE TO FIND AN ELEMENT

```
position Find (int X, List L)
{
        position P;
        P = CursorSpace [L].Next;
        while (P && CursorSpace [P].Element ! = X)
                P = CursorSpace [P].Next;
        return P;
}
```

ROUTINE TO DELETE AN ELEMENT

```
void Delete (int X, List L)
{
        position P, temp;
        P = Findprevious (X, L);
        if (! IsLast (P, L))
        {
                temp = CursorSpace [P].Next;
                CursorSpace [P].Next = CursorSpace [temp].Next;
                CursorFree (temp);
        }
}
```

ROUTINE FOR INSERTION

```
void Insert (int X, List L, position P)
{
        position newnode;
        newnode = CursorAlloc ( );
        if (newnode ! = 0)
        CursorSpace [newnode].Element = X;
        CursorSpace [newnode]. Next = CursorSpace [P].Next;
        CursorSpace [P].Next = newnode;
}
```

Example of a Cursor Implementation of Linked List

Slot	Element	Next
0	-	8
1	B	0
2	header X	4
3	C	6
4	A	1
5	header Y	3
6	D	0
7	-	0
8	-	7

The List X has A, B and list Y has C, D

Fig. 2.1.6 (a)

2.2 THE STACK ADT

2.2.1 Stack Model :

A stack is a linear data structure which follows Last In First Out (LIFO) principle, in which both insertion and deletion occur at only one end of the list called the Top.

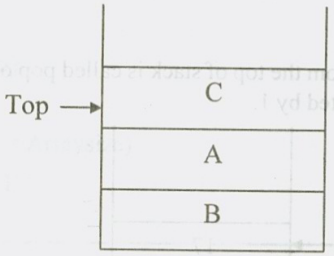


Fig. 2.2 Stack Model

Example : -

Pile of coins., a stack of trays in cafeteria.

2.2.2 Operations On Stack

The fundamental operations performed on a stack are

1. Push
2. Pop

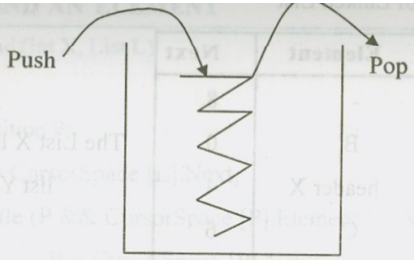


Fig. 2.2.2 Operations on stack

PUSH :

The process of inserting a new element to the top of the stack. For every push operation the top is incremented by 1.

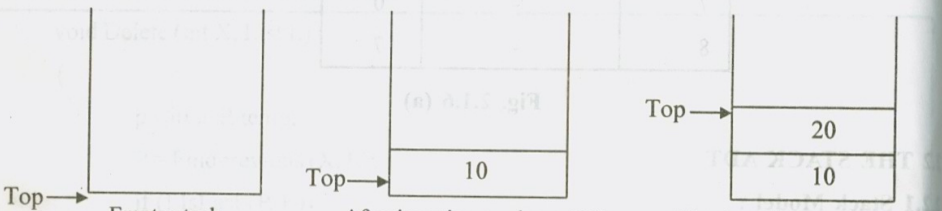


Fig. 2.2.2(a) Push Operation

POP :

The process of deleting an element from the top of stack is called pop operation. After every pop operation the top pointer is decremented by 1.

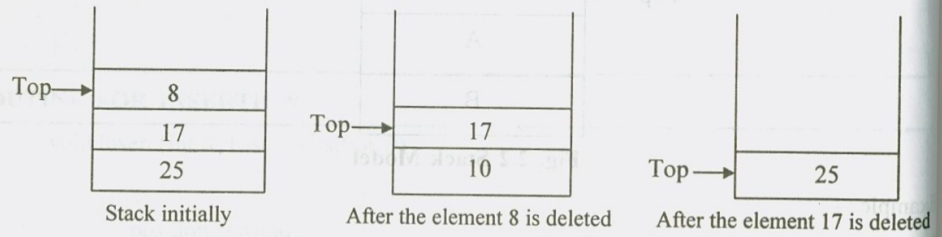


Fig.2.2.2(b) Pop operation

EXCEPTIONAL CONDITIONS

OverFlow

Attempt to insert an element when the stack is full is said to be overflow.

UnderFlow

Attempt to delete an element, when the stack is empty is said to be underflow.

2.2.3 Implementation of Stack

Stack can be implemented using arrays and pointers.

Array Implementation

In this implementation each stack is associated with a pop pointer, which is -1 for an empty stack.

- To push an element X onto the stack, Top Pointer is incremented and then set Stack [Top] = X.
- To pop an element, the stack [Top] value is returned and the top pointer is decremented.
- pop on an empty stack or push on a full stack will exceed the array bounds.

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int x, Stack S)
{
        if (IsFull (S))
                Error ("Full Stack");
        else
        {
                Top = Top + 1;
                S[Top] = X;
        }
}
int IsFull (Stack S)
{
        if (Top = = Arraysize)
         return (1);
}
```

ROUTINE TO POP AN ELEMENT FROM THE STACK

```
void pop (Stack S)
{
        if (IsEmpty (S))
                Error ("Empty Stack");
        else
        {
                X = S [Top];
                Top = Top - 1;
```

```
    }
}
int IsEmpty (Stack S)
{
    if (S→Top = = -1)
        return (1);
}
```

ROUTINE TO RETURN TOP ELEMENT OF THE STACK

```
int TopElement (Stack S)
{
    if (! IsEmpty (s))
        return S[Top];
    else
        Error ("Empty Stack");
        return 0;
}
```

LINKED LIST IMPLEMENTATION OF STACK

- Push operation is performed by inserting an element at the front of the list.
- Pop operation is performed by deleting at the front of the list.
- Top operation returns the element at the front of the list.

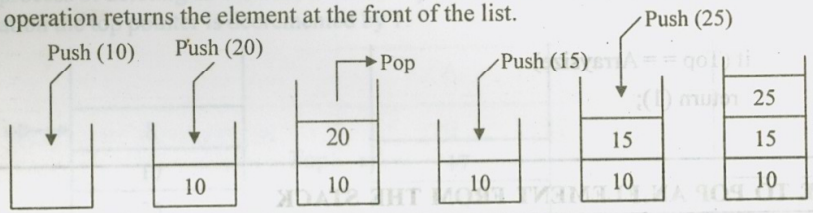


Fig. 2.2.3 Stack ADT

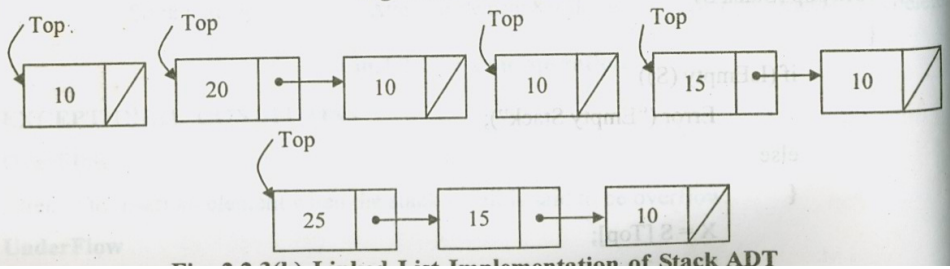


Fig. 2.2.3(b) Linked List Implementation of Stack ADT

DECLARATION FOR LINKED LIST IMPLEMENTATION

```
Struct Node;
typedef Struct Node *Stack;
int IsEmpty (Stack S);
Stack CreateStack (void);
void MakeEmpty (Stack S);
void push (int X, Stack S);
int Top (Stack S);
void pop (Stack S);
Struct Node
{
    int Element ;
    Struct Node *Next;
};
```

ROUTINE TO CHECK WHETHER THE STACK IS EMPTY

```
int IsEmpty (Stack S)
{
    if (S→Next = = NULL)
        return (1);
}
```

ROUTINE TO CREATE AN EMPTY STACK

```
Stack CreateStack ( )
{
    Stack S;
    S = malloc (Sizeof (Struct Node));
    if (S = = NULL)
        Error (" Outof Space");
    MakeEmpty (s);
    return S;
}
void MakeEmpty (Stack S)
```

```
{
        if (S = = NULL)
                Error (" Create Stack First");
        else
                while (! IsEmpty (s))
                pop (s);
}
```

ROUTINE TO PUSH AN ELEMENT ONTO A STACK

```
void push (int X, Stack S)
{
        Struct Node * Tempcell;
        Tempcell = malloc (sizeof (Struct Node));
        If (Tempcell = = NULL)
                Error ("Out of Space");
        else
        {
                Tempcell → Element = X;
                Tempcell → Next = S → Next;
                S → Next = Tempcell;
        }
}
```

ROUTINE TO RETURN TOP ELEMENT IN A STACK

```
int Top (Stack S)
{
        If (! IsEmpty (s))
                return S → Next → Element;
        Error ("Empty Stack");
        return 0;
}
```

ROUTINE TO POP FROM A STACK

```
void pop (Stack S)
{
        Struct Node *Tempcell;
        If (IsEmpty (S))
                Error ("Empty Stack");
        else
        {
                Tempcell = S → Next;
                S → Next = S → Next → Next;
                Free (Tempcell);
        }
}
```

2.2.4 APPLICATIONS OF STACK

Some of the applications of stack are :

(i) Evaluating arithmetic expression

(ii) Balancing the symbols

(iii) Towers of Hannoi

(iv) Function Calls.

(v) 8 Queen Problem.

Different Types of Notations To Represent Arithmetic Expression

There are 3 different ways of representing the algebraic expression.

They are

* INFIX NOTATION
* POSTFIX NOTATION
* PREFIX NOTATION

INFIX

In Infix notation, The arithmetic operator appears between the two operands to which it is being applied.

For example : - A / B + C

POSTFIX

The arithmetic operator appears directly after the two operands to which it applies. Also calle reverse polish notation. ((A/B) + C)

For example : - AB / C +

PREFIX

The arithmetic operator is placed before the two operands to which it applies. Also called as polis notation. ((A/B) + C)

For example : - +/ABC

	INFIX	PREFIX (or) POLISH	POSTFIX (or) REVERSE POLISH
1.	(A + B)/(C - D)	/+AB - CD	AB + CD - /
2.	A + B*(C - D)	+A*B - CD	ABCD - * +
3.	X * A / B - D	- / * XABD	X A*B/D-
4.	X + Y * (A - B) / (C - D)	+X/*Y - AB - CD	XYAB - *CD - / +
5.	A * B/C + D	+ / * ABCD	AB * C / D +

1. Evaluating Arithmetic Expression

To evaluate an arithmetic expressions, first convert the given infix expression to postfix expressio and then evaluate the postfix expression using stack.

Infix to Postfix Conversion

Read the infix expression one character at a time until it encounters the delimiter. "#"

Step 1 : If the character is an operand, place it on to the output.

Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a highe or equal priority than input operator then pop that operator from the stack and place it on the output.

Step 3 : If the character is a left paraenthesis, push it onto the stack.

Step 4 : If the character is a right paraenthesis, pop all the operators from the stack till encounters left parenthesis, discard both the parenthesis in the output.

Infix Expression : A * B + (C - D / E) #

Read Character	Stack	Output
A		A
*	*	A
B	*	AB
+		AB*
	+	AB*
(	(+	AB*
(	(+	AB* C

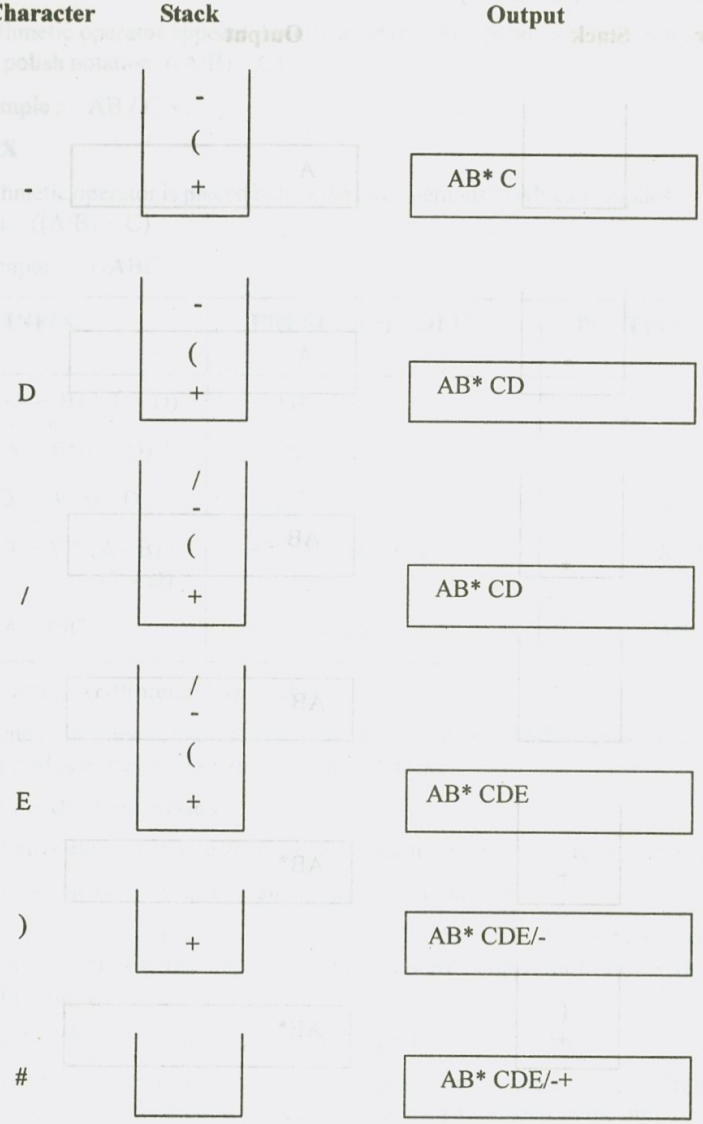


Fig. 2.2.4 (a) Conversion of Infix Expression to Postfix Expression

Evaluating Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter '#'.

Step 1 : - If the character is an operand, push its associated value onto the stack.

Step 2 : - If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

Example :

Let us consider the symbols A, B, C, D, E had the associated values :

Symbol	Value
A	4
B	5
C	5
D	8
E	2

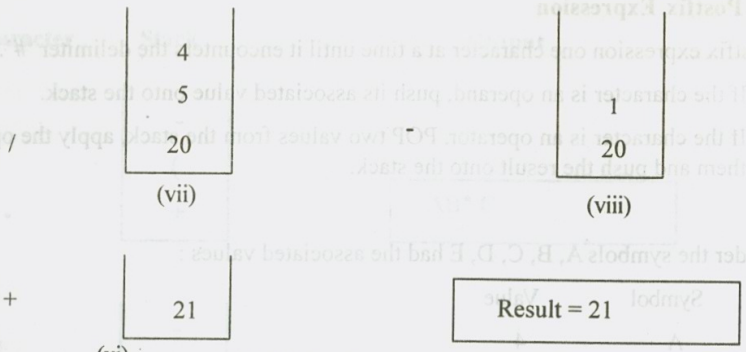


Fig. 2.2.4 (b) Evaluation of AB*CDE / - +

2. BALANCING THE SYMBOLS

Read one character at a time until it encounters the delimiter '#'.

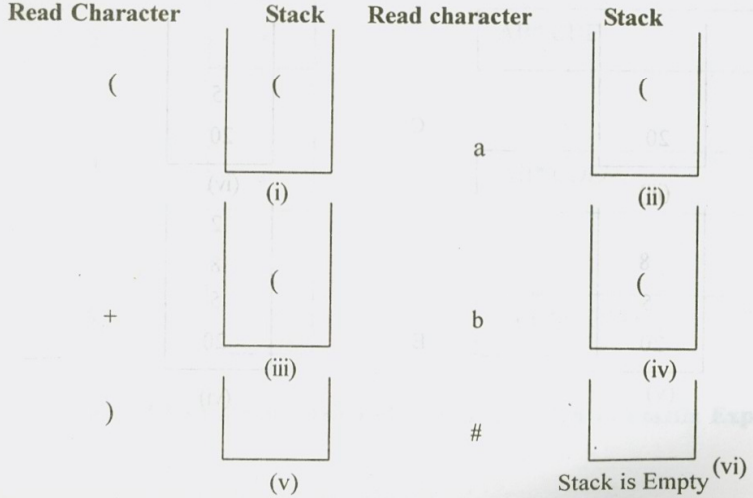
Step 1 : - If the character is an opening symbol, push it onto the stack.

Step 2 : - If the character is a closing symbol, and if the stack is empty report an error as missing opening symbol.

Step 3 : - If it is a closing symbol and if it has corresponding opening symbol in the stack, POP it from the stack. Otherwise, report an error as mismatched symbols.

Step 4 : - At the end of file, if the stack is not empty, report an error as Missing closing symbol. Otherwise, report as Balanced symbols.

Let us consider the expression as (a + b) #



Consider the expression ((a + b) # : -

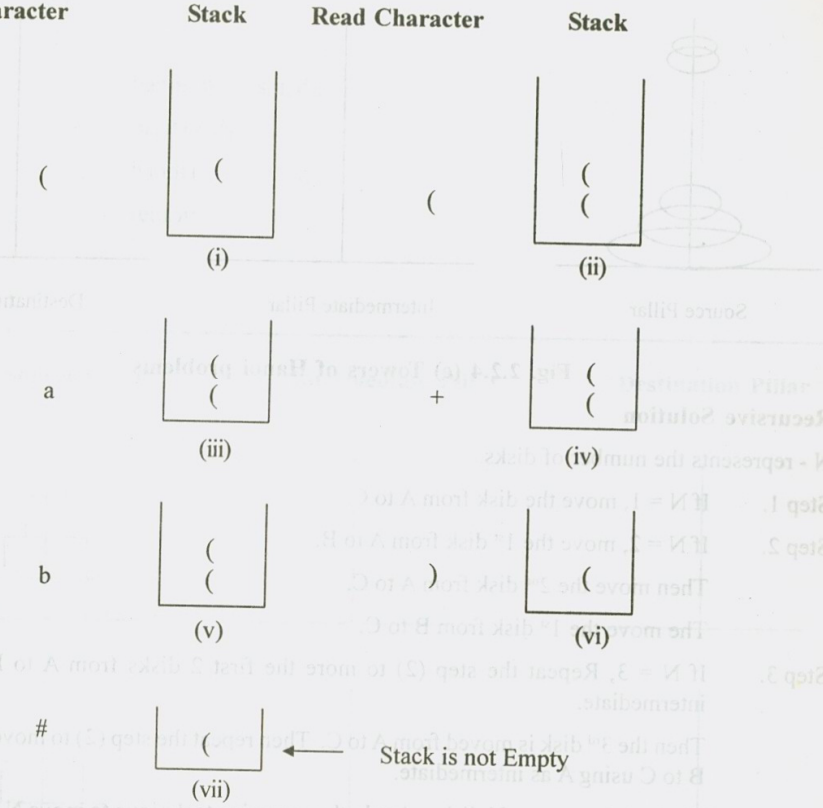


Fig. 2.2.4 (d) Illustration For Unbalanced Symbols

Towers of Hanoi

Towers of Hanoi is one of the example illustrating the Recursion technique.

The problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

Rule 1 : Only one disk could be moved at a time.

Rule 2 : No larger disk could ever reside on a pillar on top of a smaller disk.

Rule 3 : A 3rd pillar could be used as an intermediate to store one or more disks, while they were being moved from source to destination.

Fig. 2.2.4 (e) Towers of Hanoi problems .

Recursive Solution

N - represents the number of disks.

Step 1. If N = 1, move the disk from A to C.

Step 2. If N = 2, move the 1st disk from A to B.

Then move the 2nd disk from A to C,

The move the 1st disk from B to C.

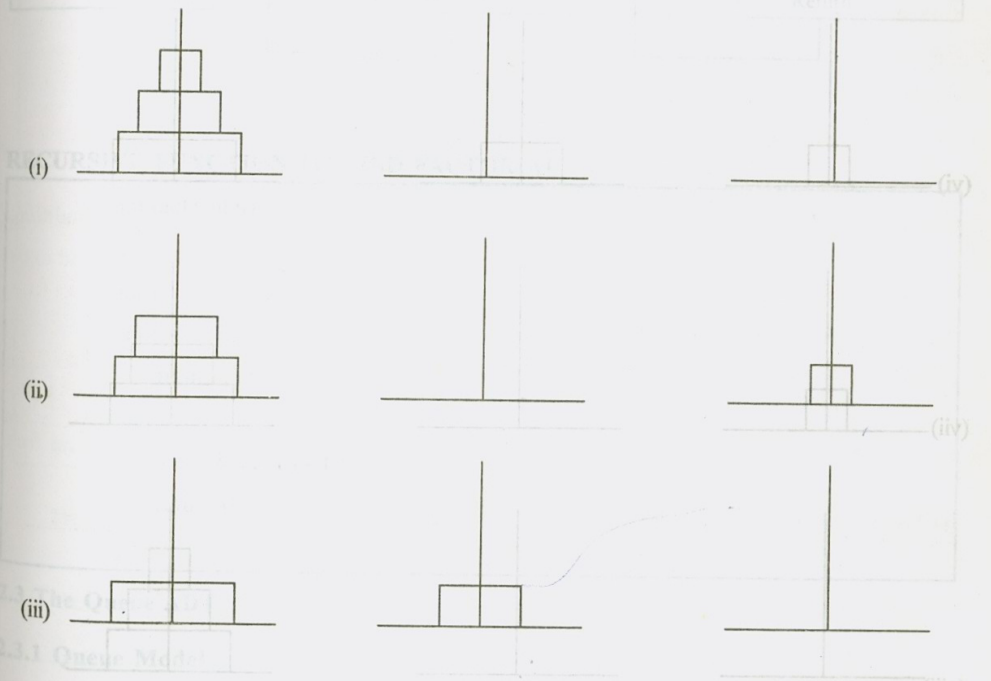
Step 3. If N = 3, Repeat the step (2) to more the first 2 disks from A to B using C as intermediate.

Then the 3rd disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as intermediate.

In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the Nth disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate.

RECURSIVE ROUTINE FOR TOWERS OF HANOI

```
void hanoi (int n, char s, char d, char i)
{
    /* n ⇒ no. of disks, s → source, d → destination i → intermediate */
    if (n == 1)
    {
        print (s, d);
    }
    else
    {
        hanoi (n - 1, s, i, d);
        print (s, d)
        hanoi (n-1, i, d, s);
        return;
    }
}
```



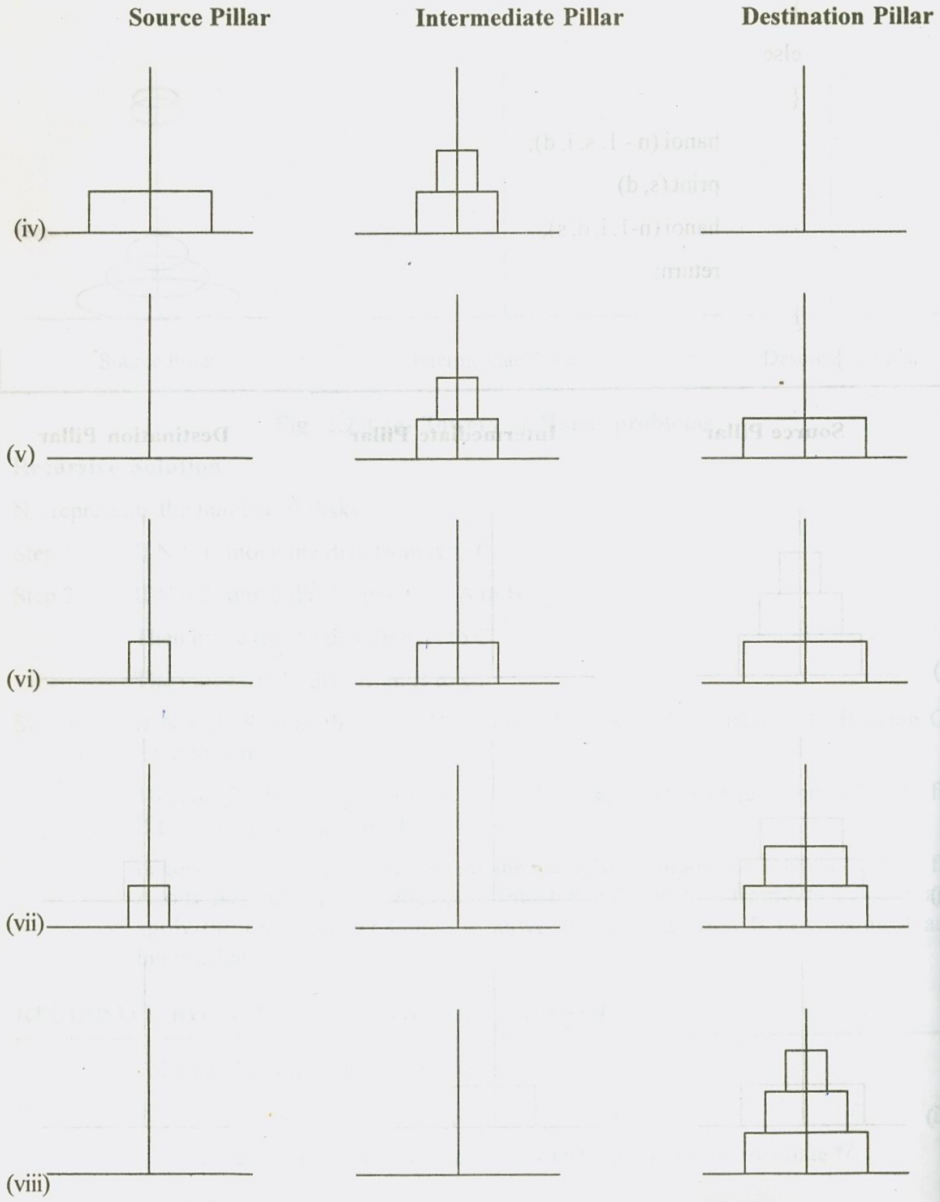
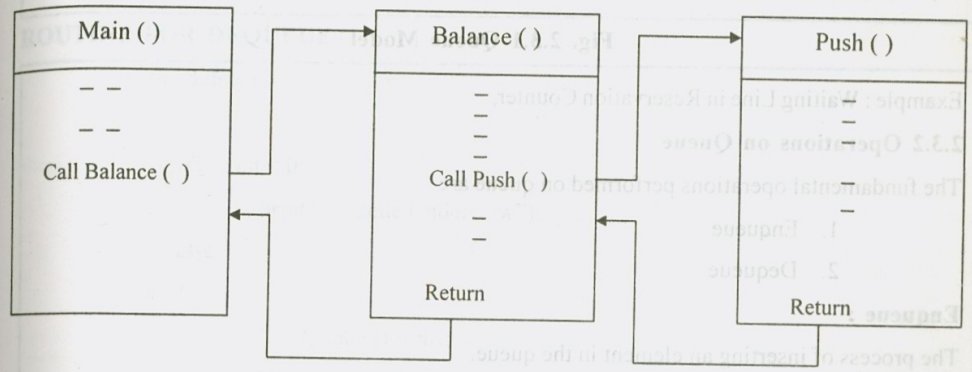


Fig. 2.2.4 (f) Towers of Hanoi Problem for 3 Disks

4. Function Calls

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.



RECURSIVE FUNCTION TO FIND FACTORIAL : -

```
int fact (int n)
{
int s;
if (n = = 1)
    return (1);
else
    s = n * fact (n - 1);
    return (s);
}
```

2.3 The Queue ADT

2.3.1 Queue Model

A Queue is a linear data structure which follows First In First Out (FIFO) principle, in which insertion is performed at rear end and deletion is performed at front end.

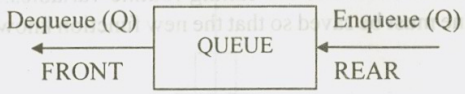


Fig. 2.3.1 Queue Model

Example : Waiting Line in Reservation Counter,

2.3.2 Operations on Queue

The fundamental operations performed on queue are

1. Enqueue
2. Dequeue

Enqueue :

The process of inserting an element in the queue.

Dequeue :

The process of deleting an element from the queue.

Exception Conditions

Overflow : Attempt to insert an element, when the queue is full is said to be overflow condition.

Underflow : Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

2.3.3 Implementation of Queue

Queue can be implemented using arrays and pointers.

Array Implementation

In this implementation queue Q is associated with two pointers namely rear pointer and front pointer.

To insert an element X onto the Queue Q, the rear pointer is incremented by 1 and then set

Queue [Rear] = X

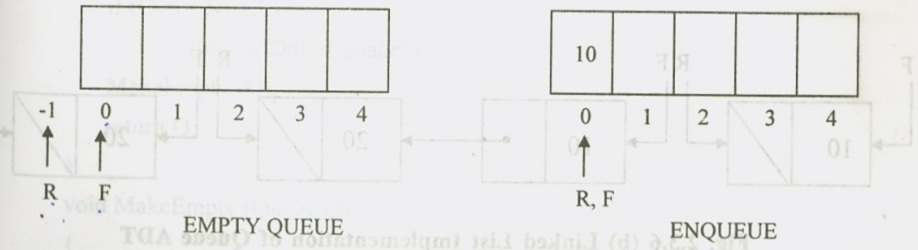
To delete an element, the Queue [Front] is returned and the Front Pointer is incremented by 1.

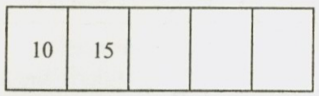
ROUTINE TO ENQUEUE

```
void Enqueue (int X)
{
    if (rear > = max _ Arraysize)
        print (" Queue overflow");
    {
        Rear = Rear + 1;
        Queue [Rear] = X;
    }
}
```

ROUTINE FOR DEQUEUE

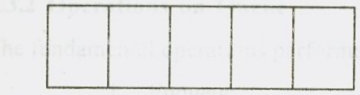
```
void delete ( )
{
    if (Front < 0)
        print (" Queue Underflow");
    else
    {
        X = Queue [Front];
        if (Front = = Rear)
        {
            Front = 0;
            Rear = -1;
        }
        else
            Front = Front + 1 ;
    }
}
```





ENQUEUE (15)

DEQUEUE (Q)



DEQUEUE (Q)

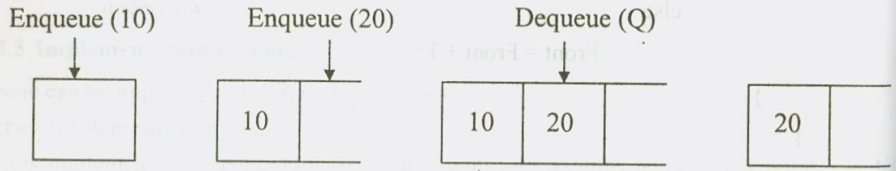
In Dequeue operation, if Front = Rear, then reset both the pointers to their initial values. (i.e. F = 0, R = -1)

Fig. 2.3.3 (a) Illustration for Array Implementation of Queue.

Linked List Implementation of Queue

Enqueue operation is performed at the end of the list.

Dequeue operation is performed at the front of the list.



Queue ADT

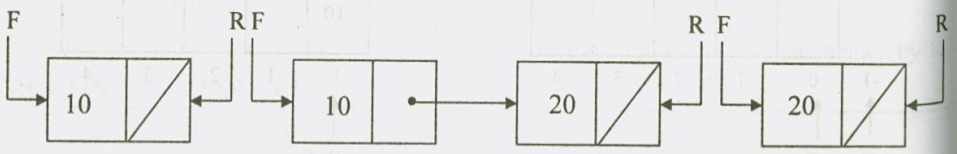


Fig. 2.3.6 (b) Linked List Implementation of Queue ADT

DECLARATION FOR LINKED LIST IMPLEMENTATION OF QUEUE ADT

```
Struct Node;
typedef Struct Node * Queue;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue Q);
Struct Node
{
    int Element;
    Struct Node *Next;
}* Front = NULL, *Rear = NULL;
```

ROUTINE TO CHECK WHETHER THE QUEUE IS EMPTY

```
int IsEmpty (Queue Q)                    // returns boolean value /
{                                        // if Q is empty
    if (Q→Next = = NULL)                 // else returns 0
      return (1);
}
```

ROUTINE TO CHECK AN EMPTY QUEUE

```
Struct CreateQueue ( )
{
    Queue Q;
    Q = Malloc (Sizeof (Struct Node));
    if (Q = = NULL)
            Error ("Out of Space");
    MakeEmpty (Q);
    return Q;
}
void MakeEmpty (Queue Q)
{
```

```
            Error ("Create Queue First");
        else
            while (! IsEmpty (Q)
                Dequeue (Q);
}
```

ROUTINE TO ENQUEUE AN ELEMENT IN QUEUE

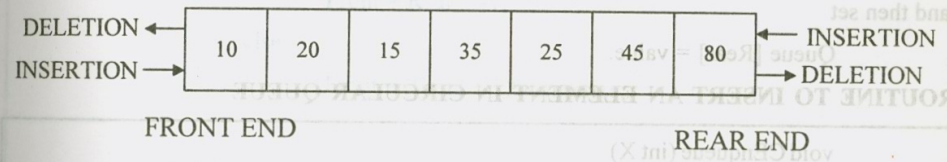
```
void Enqueue (int X)
{
    Struct node *newnode;
    newnode = Malloc (sizeof (Struct node));
    if (Rear = = NULL)
    {
        newnode →data = X;
        newnode →Next = NULL;
        Front = newnode;
        Rear = newnode;
    }
    else
    {
        newnode →data = X;
        newnode →Next = NULL;
        Rear →next = newnode;
        Rear = newnode;
    }
}
```

ROUTINE TO DEQUEUE AN ELEMENT FROM THE QUEUE

```
void Dequeue ( )
{
    Struct node *temp;
    if (Front = = NULL)
        Error("Queue is underflow");
    else
    {
        temp = Front;
        {
            Front = NULL;
            Rear = NULL;
        }
        else
            Front = Front →Next;
            Print (temp →data);
            free (temp);
    }
}
```

2.3.4 Double Ended Queue (DEQUE)

In Double Ended Queue, insertion and deletion operations are performed at both the ends.

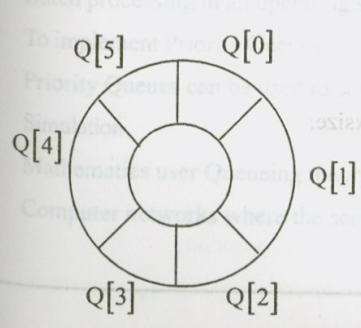


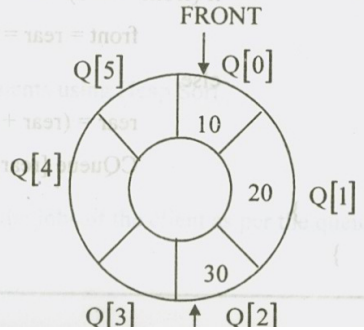
2.3.5 Circular Queue

In Circular Queue, the insertion of a new element is performed at the very first location of the queue if the last location of the queue is full, in which the first element comes just after the last element.

Advantages

It overcomes the problem of unutilized space in linear queues, when it is implemented as arrays.





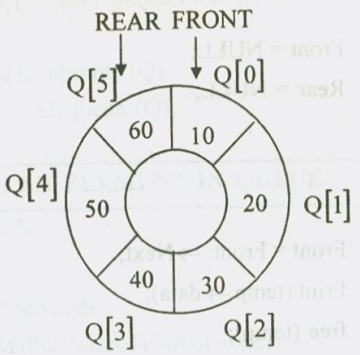


Fig. 2.3.5 Insertion in a Circular Queue

To perform the insertion of an element to the queue, the position of the element is calculated by the relation as

Rear = (Rear + 1) % Maxsize.

and then set

Queue [Rear] = value.

ROUTINE TO INSERT AN ELEMENT IN CIRCULAR QUEUE

```
void CEnqueue (int X)
{
    if (Front = = (rear + 1) % Maxsize)
            print ("Queue is overflow");
    else
    {
            if (front = = -1)
                    front = rear = 0;
            else
                    rear = (rear + 1)% Maxsize;
                    CQueue [rear] = X;
    }
}
```

To perform the deletion, the position of the Front printer is calculated by the relation

Value = CQueue [Front]

Front = (Front + 1) % maxsize.

ROUTINE TO DELETE AN ELEMENT FROM CIRCULAR QUEUE

```
int CDequeue ( )
{
    if (front = = -1)
            print ("Queue is underflow");
    else
    {
            X = CQueue [Front];
            if (Front = = Rear)
                    Front = Rear = -1;
            else
                    Front = (Front + 1)% maxsize;
    }
    return (X);
}
```

2.3.6 Priority Queues

Priority Queue is a Queue in which inserting an item or removing an item can be performed from any position based on some priority.

2.3.7 Applications of Queue

* Batch processing in an operating system
* To implement Priority Queues.
* Priority Queues can be used to sort the elements using Heap Sort.
* Simulation.
* Mathematics user Queueing theory.
* Computer networks where the server takes the jobs of the client as per the queue strategy.

Questions

Part - A

1. Define ADT.
2. What are the advantages of linked list over arrays?
3. What are the advantages of doubly linked list over singly linked list?
4. List the applications of List ADT.
5. Write a procedure for polynomial differentiation.
6. What are the operations performed on stack and write its exceptional condition?
7. What do you mean by cursor implementation of list?
8. List the application of stack
9. Convert the infix expression a + b * c + (d * e + f) * g to its equivalent postfix expression and prefix expression.
10. Convert the infix expression (a * b) + ((c * g) - (e / f)) to its equivalent polish and reverse polish expression.
11. Write the recursive routine to perform factorial of a given number.
12. Define Queue data structure and give some applications for it.
13. What is Deque?
14. What is Circular Queue?
15. What is Priority Queue?
16. Write a routine to return the top element of stack.
17. Write a routine to check IsEmpty and Islast for queue.
18. Write a procedure to insert an element in a singly linked list
19. What is the principle of radix sort?
20. What is Multilist?

Part - B

1. Explain the array and linked list implementation of stack.
2. Explain the array and linked list implementation of Queue.
3. What are the various linked list operations? Explain
4. Explain how stack is applied for evaluating an arithemetic expression.
5. Write routines to implement addition, subtraction & differentiation of two polynomials.
7. Explain Cursor implementation of List?
8. Write the operations performed on singly linked list?
9. Write the insertion and deletion routine for doubly linked list?
10. Write the procedure for polynomial addition and differentiation?

INTRODUCTION TO DATA STRUCTURES

As computers become faster and faster, the need for programs that can handle large amounts of input becomes more acute which inturn requires choosing suitable algorithm with more efficiency, therefore the study of data structures because an important task which describes, methods of organizing large amounts of data and manipulating them in a better way.

DATA

A collection of facts, concepts, figures, observation, occurrences or instructions in a formalized manner.

INFORMATION

The meaning that is currently assigned to data by means of the conventions applied to those data (i.e processed data).

RECORD

Collection of related fields.

DATATYPE

Set of elements that share common set of properties used to solve a program.

DATASTRUCTURE

A way of organizing, storing and retrieving data and their relationship with each other.

ALGORITHM

An algorithm is a logical module designed to handle a specific problem relative to a particular data structure.

DESIRABLE PROPERTIES OF AN EFFECTIVE ALGORITHM

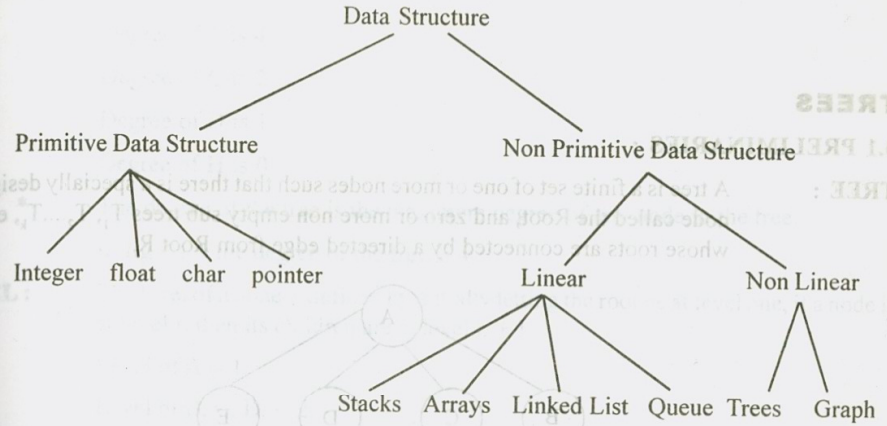
* It should be clear / complete and definite.
* It should be efficient.
* It should be concise and compact.
* It should be less time consuming.
* Effective memory utilization.

APPLICATIONS OF DATA STRUCTURES

Some of the applications of data structures are :

* Operating systems
* Compiler design
* Statistical and Numerical analysis
* Database Management systems
* Expert systems

CLASSIFICATION OF DATA STRUCTURE



Primitive Data Structure

It is a basic data structure which can be directly operated by the machine instruction.

Non Primitive Data Structure

Data Structures emphasize on structuring of a group of homogeneous or hetrogeneous data items.

Linear Data Structure

A data structure which contains a linear arrangement of elements in the memory.

Non - Linear Data Structure

A data structure which represents a hierarchical arrangement of elements.

3 TREES

3.1 PRELIMINARIES :

TREE : A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T_1, T_2....T_k, each of whose roots are connected by a directed edge from Root R.

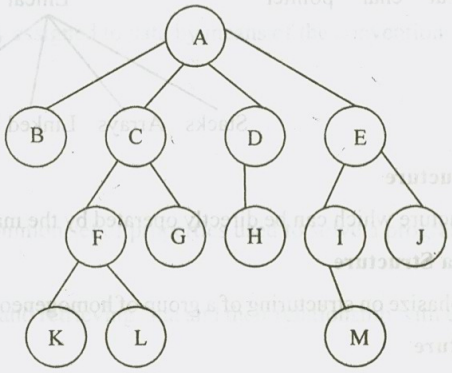


Fig. 3.1.1 Tree

ROOT : A node which doesn't have a parent. In the above tree. The Root is A.

NODE : Item of Information.

LEAF : A node which doesn't have children is called leaf or Terminal node. Here B, K, L, G, H, M, J are leafs.

SIBLINGS : Children of the same parents are said to be siblings, Here B, C, D, E are siblings, F, G are siblings. Similarly I, J & K, L are siblings.

PATH : A path from node n, to n_k is defined as a sequence of nodes n_1, n_2,n_3....n_k such that n_i is the parent of n_{i+1}. for $1 \le i < k$. There is exactly only one path from each node to root.

In fig 3.1.1 path from A to L is A, C, F, L. where A is the parent for C, C is the parent of F and F is the parent of L.

LENGTH : The length is defined as the number of edges on the path.

In fig 3.1.1 the length for the path A to L is 3.

DEGREE : The number of subtrees of a node is called its degree.

In fig 3.1.1

Degree of A is 4

Degree of C is 2

Degree of D is 1

Degree of H is 0.

* The degree of the tree is the maximum degree of any node in the tree.

In fig 3.1.1 the degree of the tree is 4.

LEVEL : The level of a node is defined by initially letting the root be at level one, if a node is at level L then its children are at level L + 1.

Level of A is 1.

Level of B, C, D, is 2.

Level of F, G, H, I, J is 3

Level of K, L, M is 4.

DEPTH : For any node n, the depth of n is the length of the unique path from root to n.

The depth of the root is zero.

In fig 3.1.1 Depth of node F is 2.

Depth of node L is 3.

HEIGHT : For any node n, the height of the node n is the length of the longest path from n to the leaf.

The height of the leaf is zero

In fig 3.1.1 Height of node F is 1.

Height of L is 0.

Note : The height of the tree is equal to the height of the root

Depth of the tree is equal to the height of the tree.

3.2 BINARY TREE

Definition :-

Binary Tree is a tree in which no node can have more than two children.

Maximum number of nodes at level i of a binary tree is 2^{i-1}.

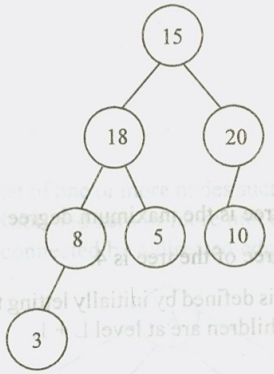
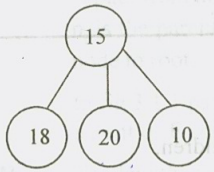
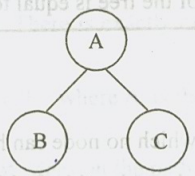


Fig. 3.2.1 Binary Tree

BINARY TREE NODE DECLARATIONS

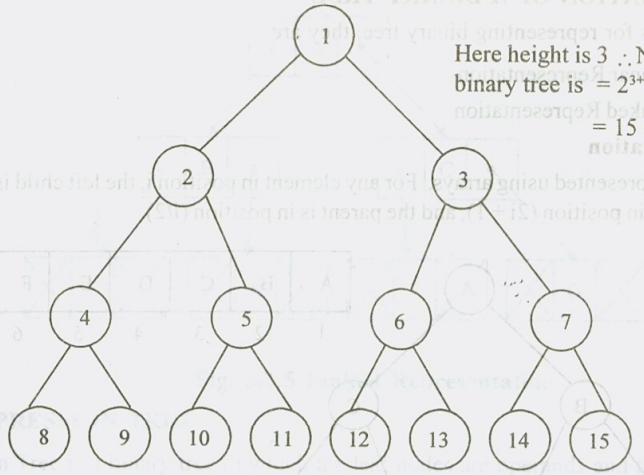
```
Struct TreeNode
{
        int Element;
        Struct TreeNode *Left ;
        Struct TreeNode *Right;
};
```

COMPARISON BETWEEN GENERAL TREE & BINARY TREE

General Tree	Binary Tree
* General Tree has any number of children.	* A Binary Tree has not more than two children.
	

FULL BINARY TREE :-

A full binary tree of height h has 2^{h+1} - 1 nodes.



Here height is 3 ∴ No. of nodes in full binary tree is $= 2^{3+1} - 1$
$= 15$ nodes.

Fig. 3.2.2 A Full Binary Tree

COMPLETE BINARY TREE :

A complete binary tree of height h has between 2^h and 2^{h+1} - 1 nodes. In the bottom level the elements should be filled from left to right.

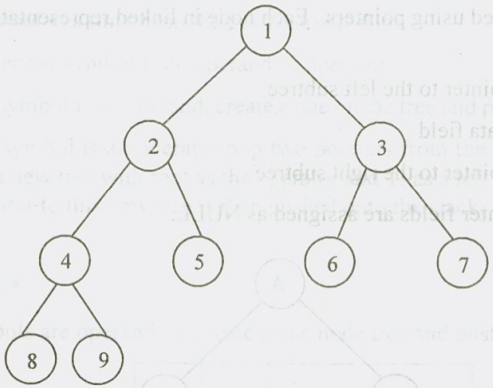


Fig. 3.2.3 A Complete Binary Tree.

Note : A full binary tree can be a complete binary tree, but all complete binary tree is not a full binary tree.

3.2.1REPRESENTATION OF A BINARY TREE

There are two ways for representing binary tree, they are

* Linear Representation
* Linked Representation

Linear Representation

The elements are represented using arrays. For any element in position i, the left child is in position 2i, the right child is in position (2i + 1), and the parent is in position (i/2).

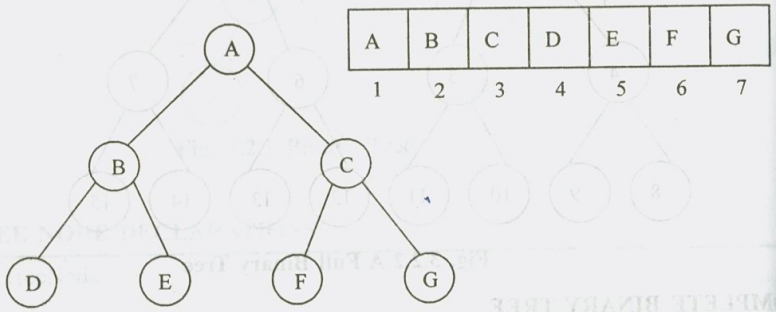


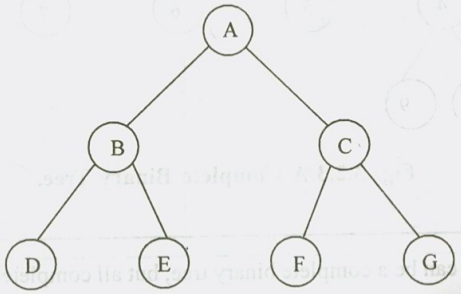
Fig. 3.2.4 Linear Representation

Linked Representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

* Pointer to the left subtree
* Data field
* Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.



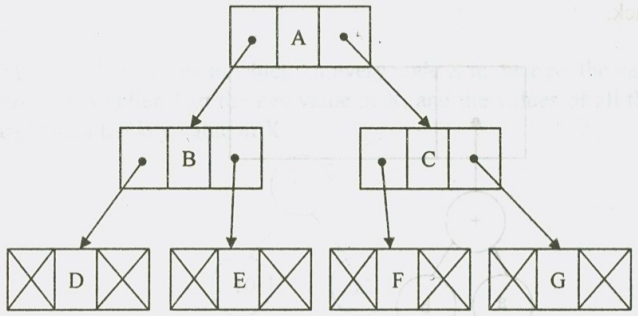


Fig. 3.2.5 Linked Representation

3.2.2 EXPRESSION TREE

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be travesed by inorder, preorder and postorder traversal.

Constructing an Expression Tree

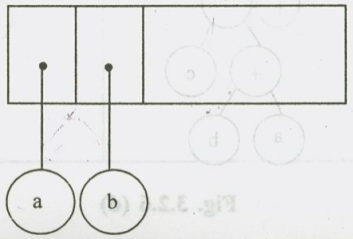
Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
   (a) If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
   (b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new tree with root as the operator and T_2 as a left child and T_1 as a right child. A pointer to this new tree is then pushed onto the stack.

Example : -

ab + c *

The first two symbols are operand, so create a one node tree and push the pointer on to the stack.



Next '+' symbol is read, so two pointers are popped, a new tree is formed and a pointer to this is pushed on to the stack.

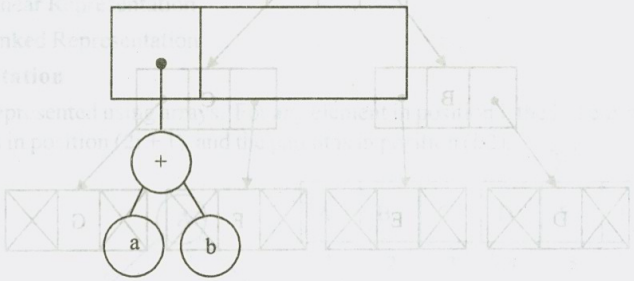


Fig. 3.2.6 (b)

Next the operand C is read, so a one node tree is created and the pointer to it is pushed onto the stack.

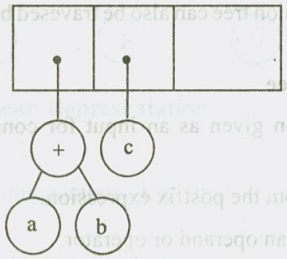


Fig. 3.2.6 (c)

Now '*' is read, so two trees are merged and the pointer to the final tree is pushed onto the stack.

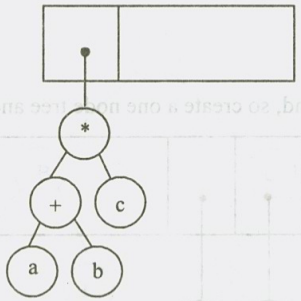


Fig. 3.2.6 (d)

3.3 The Search Tree ADT : - Binary Search Tree

Definition : -

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

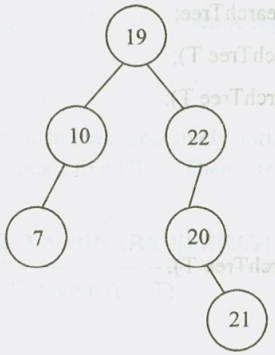
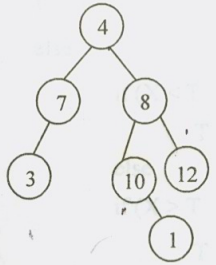
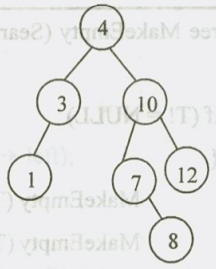


Fig. 3.3.1 Binary Search Tree

Comparision Between Binary Tree & Binary Search Tree

Binary Tree	Binary Search Tree
* A tree is said to be a binary tree if it has atmost two childrens.	* A binary search tree is a binary tree in which the key values in the left node is less than the root and the keyvalues in the right node is greater than the root.
* It doesn't have any order.	
* Example 	

Note : * Every binary search tree is a binary tree.

* All binary trees need not be a binary search tree.

DECLARATION ROUTINE FOR BINARY SEARCH TREE

```
Struct TreeNode;
typedef struct TreeNode   * SearchTree;
SearchTree Insert (int X, SearchTree T);
SearchTree Delete (int X, SearchTree T);
int Find (int X, SearchTree T);
int FindMin (Search Tree T);
int FindMax (SearchTree T);
SearchTree MakeEmpty (SearchTree T);
Struct TreeNode
{
        int Element ;
        SearchTree Left;
        SearchTree Right;
};
```

Make Empty :-

This operation is mainly for initialization when the programmer prefer to initialize the first element as a one - node tree.

ROUTINE TO MAKE AN EMPTY TREE :-

```
SearchTree MakeEmpty (SearchTree T)
{
        if (T! = NULL)
        {
                MakeEmpty (T → left);
                MakeEmpty (T → Right);
                free (T);
        }
        return NULL ;
}
```

Insert : -

To insert the element X into the tree,

* Check with the root node T
* If it is less than the root,

  Traverse the left subtree recursively until it reaches the T → left equals to NULL. Then X is placed in T → left.

* If X is greater than the root.

  Traverse the right subtree recursively until it reaches the T → right equals to NULL. Then x is placed in T → Right.

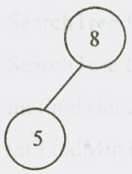
ROUTINE TO INSERT INTO A BINARY SEARCH TREE

```
SearchTree Insert (int X, searchTree T)
{
        if (T = = NULL)
        {
                T = malloc (size of (Struct TreeNode));
                if (T! = NULL) // First element is placed in the root.
                {
                        T → Element = X;
                        T → left    = NULL;
                        T → Right   = NULL;
                }
        }
        else
                if (X < T → Element)
                        T → left = Insert (X, T → left);
                else
                if (X > T → Element)
                        T → Right = Insert (X, T → Right);
        // Else X is in the tree already.
        return T;
}
```

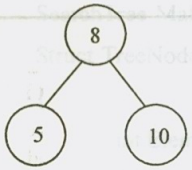
Example : -

To insert 8, 5, 10, 15, 20, 18, 3

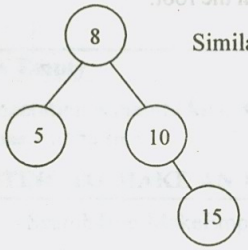
* First element 8 is considered as Root.



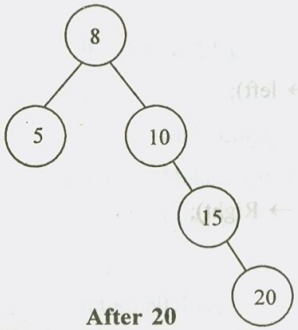
As 5 < 8, Traverse towards left



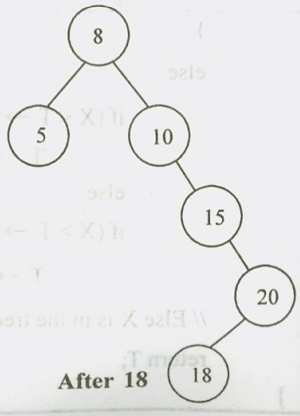
10 > 8, Traverse towards Right.



Similarly the rest of the elements are traversed.



After 20



After 18

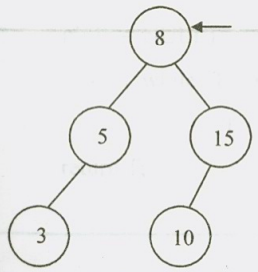
Find : -

* Check whether the root is NULL if so then return NULL.
* Otherwise, Check the value X with the root node value (i.e. T → data)
  - (1) If X is equal to T → data, return T.
  - (2) If X is less than T → data, Traverse the left of T recursively.
  - (3) If X is greater than T → data, traverse the right of T recursively.

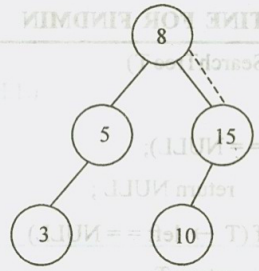
ROUTINE FOR FIND OPERATION

```
Int Find (int X, SearchTree T)
{
        If T = = NULL)
                Return NULL ;
        If (X < T → Element)
                return Find (X, T → left);
        else
        If (X > T → Element)
                return Find (X, T → Right);
        else
                return T; // returns the position of the search element.
}
```

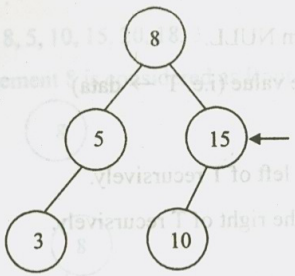
Example : - To Find an element 10 (consider, X = 10)



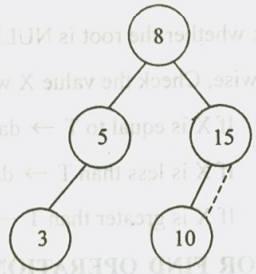
10 is checked with the Root



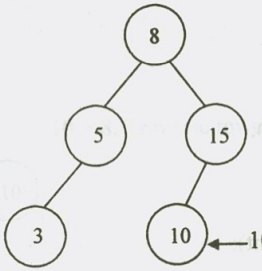
10 > 8, Go to the right child of 8



10 is checked with Root 15



10 < 15, Go to the left child of 15.



10 is checked with root 10 (Found)

Find Min :

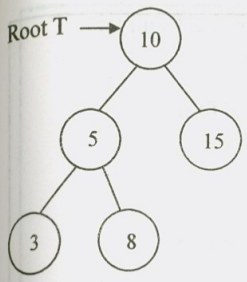
This operation returns the position of the smallest element in the tree.

To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

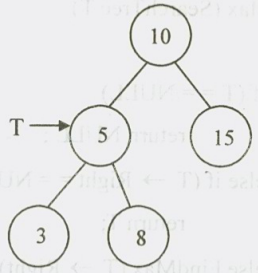
RECURISVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)
{
        if (T = = NULL);
                return NULL ;
        else if (T → left = = NULL)
                return T;
        else
                return FindMin (T → left);
}
```

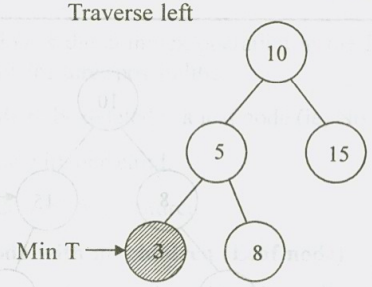
Example : -



(a) T! = NULL and T → left!=NULL, Traverse left



(b) T! = NULL and T → left!=NULL, Traverse left



(c) Since T → left is Null, return T as a minimum element.

NON - RECURSIVE ROUTINE FOR FINDMIN

```
int FindMin (SearchTree T)
{
        if (T! = NULL)
                while (T → Left ! = NULL)
                        T = T → Left ;
        return T;
}
```

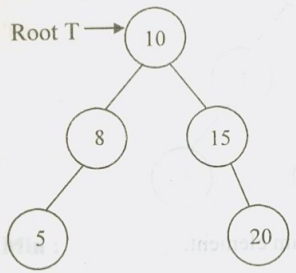
FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

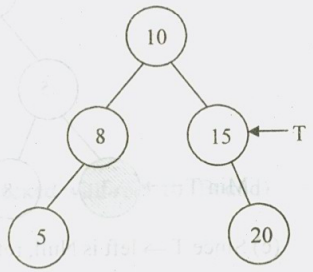
RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)
{
        if (T = = NULL)
                return NULL ;
        else if (T → Right = = NULL)
                return T;
        else FindMax (T → Right);
}
```

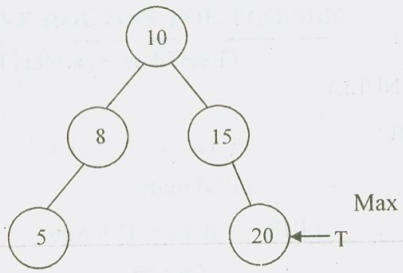
Example :-



(a) T! = NULL and T → Right!=NULL, Traverse Right



(b) T! = NULL and T → Right!=NULL, Traverse Right



(c) Since T → Right is NULL, return T as a Maximum element.

NON - RECURSIVE ROUTINE FOR FINDMAX

```
int FindMax (SearchTree T)
{
        if (T! = NULL)
                while (T → Right ! = NULL)
                        T = T → Right ;
        return T ;
}
```

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1 → Node to be deleted is a leaf node (ie) No children.

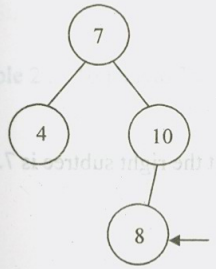
CASE 2 → Node with one child.

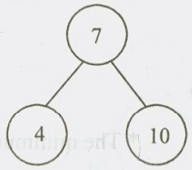
CASE 3 → Node with two children.

CASE 1 → Node with no children (Leaf node)

If the node is a leaf node, it can be deleted immediately.

Delete : 8



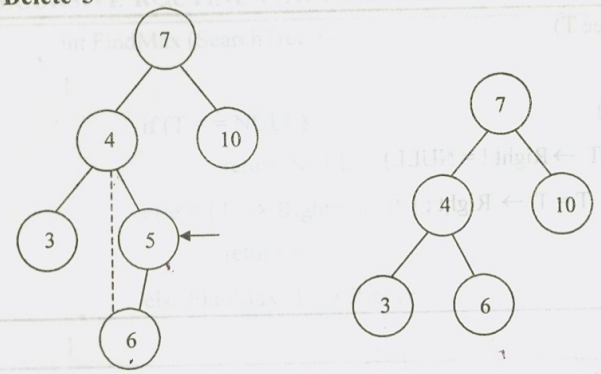


After the deletion

CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5



before deletion

After deletion

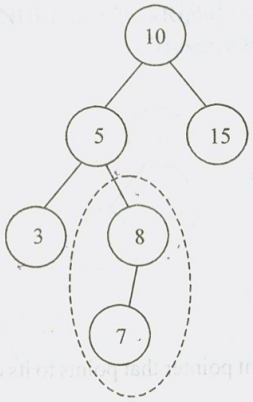
To delete 5, the pointer currently pointing the node 5 is now made to to its child node 6.

Case 3 : Node with two children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

Example 1 :

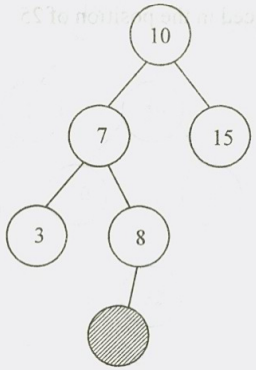
To Delete 5 :



(a)

* The minimum element at the right subtree is 7.

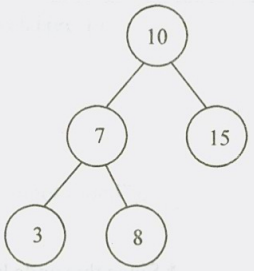
* Now the value 7 is replaced in the position of 5.



(b)

* Since the position of 7 is the leaf node delete immediately.

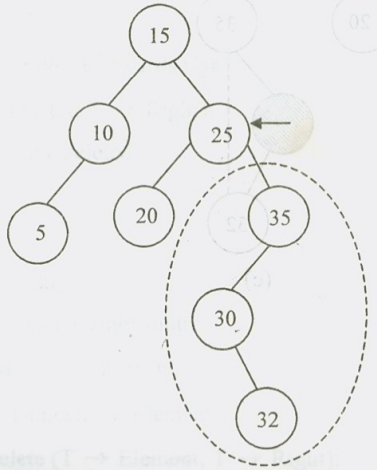
After deleting the node 5



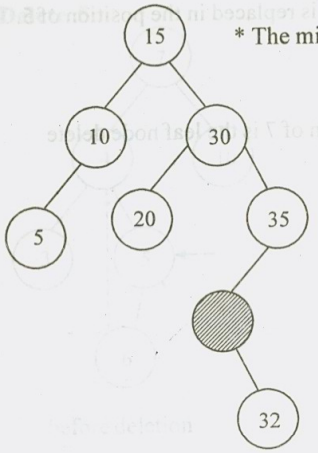
(c)

Example 2 : - To Delete 25

* The minimum element at the right subtree of 25 is 30

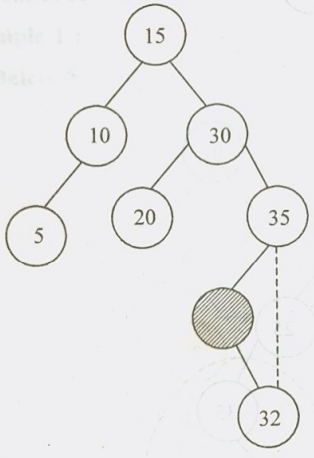


(a)



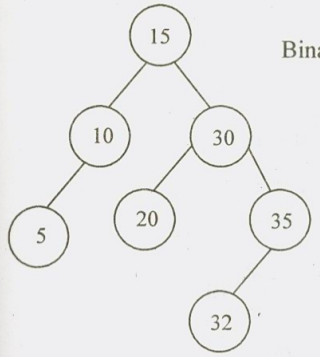
(b)

* The minimum value 30 is replaced in the position of 25



(c)

* Since this node has one child, the pointer currently pointing to this node is made to points to its child node 32.



(d)

Binary Search Tree after deleting 25

DELETION ROUTINE FOR BINARY SEARCH TREES

```
SearchTree Delete (int X, searchTree T)
{
    int Tmpcell ;
    if (T = = NULL)
        Error ("Element not found");
    else
    if (X < T → Element) // Traverse towards left
        T → Left = Delete (X, T→Left);
    else
    if (X > T → Element) // Traverse towards right
        T → Right = Delete (X, T → Right);
        // Found Element tobe deleted
    else
        // Two children
    if (T → Left && T → Right)
    { // Replace with smallest data in right subtree
        Tmpcell = FindMin (T → Right);
        T → Element = Tmpcell → Element ;
        T → Right = Delete (T → Element; T → Right);
    }
```

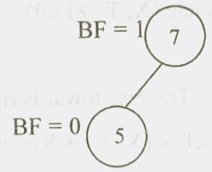
```
else // one or zero children
{
        Tmpcell = T;
        if (T → Left = = NULL)
                T = T → Right;
        else if (T → Right = = NULL)
                T = T → Left ;
        free (TmpCell);
}
return T;
}
```

3.4 AVL Tree : - (Adelson - Velskill and LANDIS)

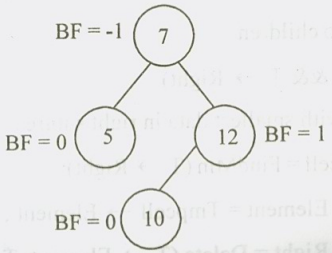
An AVL tree is a binary search tree except that for every node in the tree, the height of the left and right subtrees can differ by atmost 1.

The height of the empty tree is defined to be - 1.

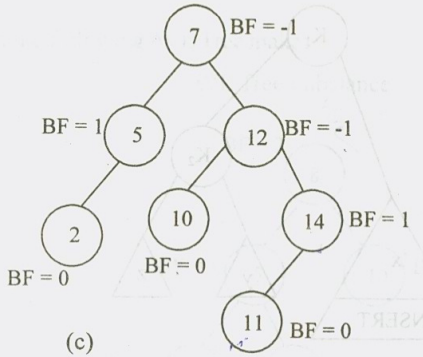
A balance factor is the height of the left subtree minus height of the right subtree. For an AVL tree all balance factor should be +1, 0, or -1. If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making either single or double rotations.



(a)



(b)



(c)

Fig. 3.4.1 AVL Tree

An AVL tree causes imbalance, when any one of the following conditions occur.

Case 1 : An insertion into the left subtree of the left child of node α.

Case 2 : An insertion into the right subtree of the left child of node α.

Case 3 : An insertion into the left subtree of the right child of node α.

Case 4 : An insertion into the right subtree of the right child of node α.

These imbalances can be overcome by

1. Single Rotation
2. Double Rotation.

Single Rotation

Single Rotation is performed to fix case 1 and case 4.

Case 1. An insertion into the left subtree of the left child of K_2.

Single Rotation to fix Case 1.

General Representation

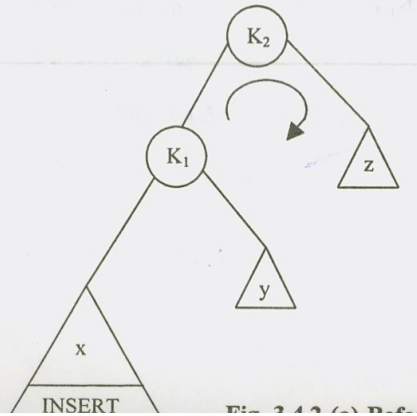


Fig. 3.4.2 (a) Before rotation

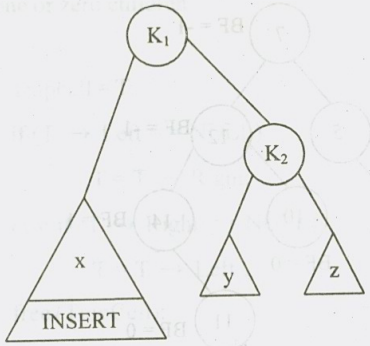


Fig. 3.4.2 (b) After rotation

ROUTINE TO PERFORM SINGLE ROTATION WITH LEFT

```
SingleRotatewithLeft (Position K2)
{
    Position K1;
    K1 = K2 → Left ;
    K2 → left = K1 → Right ;
    K1 → Right = K2 ;
    K2 → Height = Max (Height (K2 → Left), Height (K2 → Right)) + 1 ;
    K1 → Height = Max (Height (K1 → left), Height (K1 → Right)) + 1;
    return K1 ;
}
```

Example :

Inserting the value '1' in the following AVL Tree makes

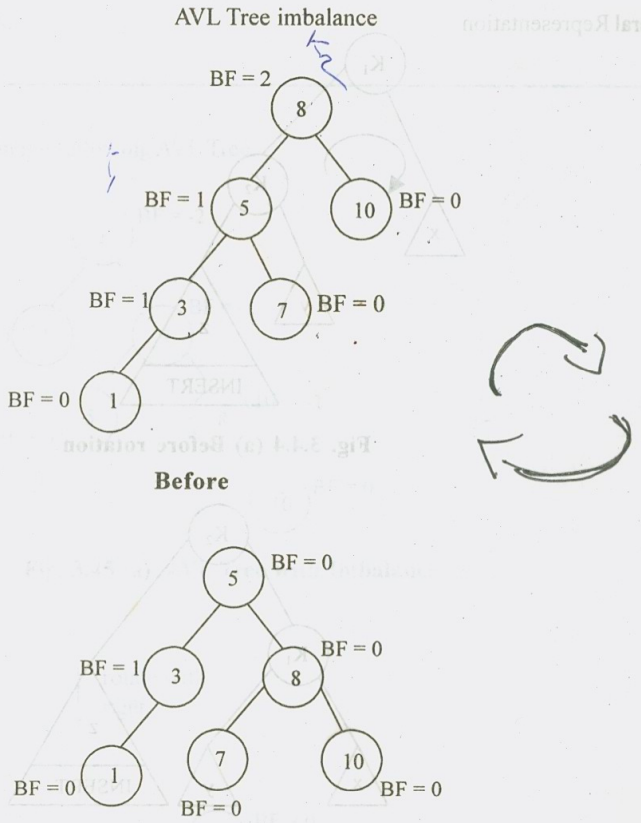


Fig. 3.4.3

Single Rotation to fix Case 4 :-

Case 4 : - An insertion into the right subtree of the right child of K_1.

General Representation

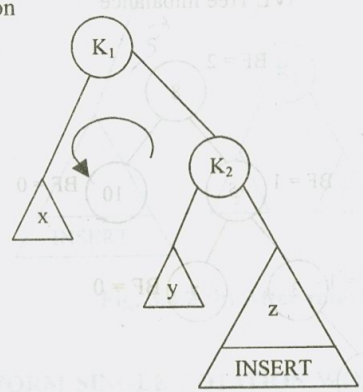


Fig. 3.4.4 (a) Before rotation

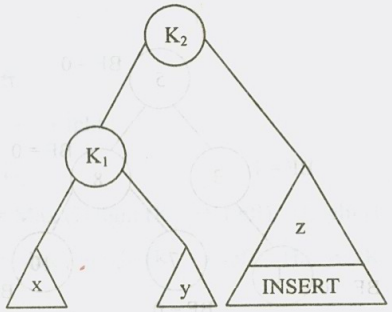


Fig. 3.4.4 (b) After Rotation

ROUTINE TO PERFORM SINGLE ROTATION WITH RIGHT :-

```
Single Rotation With Right (Position K2)
{
    Position K2 ;
    K2 = K1 → Right;
    K1 → Right = K2 → Left ;
    K2 → Left = K1 ;
    K2 → Height = Max (Height (K2 → Left), Height (K2 → Right)) +1 ;
    K1 → Height = Max (Height (K1 → Left), Height (K1 → Right)) +1 ;
    Return K2 ;
}
```

example : -

inserting the value '10' in the following AVL Tree.

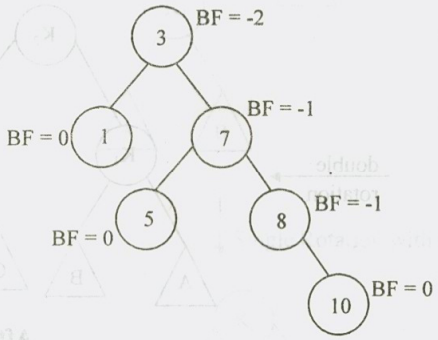


Fig. 3.45 (a) AVL Tree with Imbalance

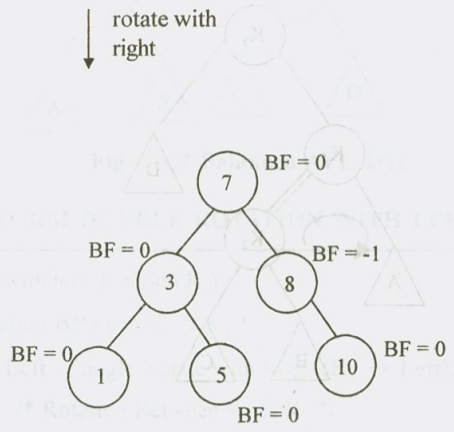


Fig. 3.4.5 (b) Balanced AVL Tree

Double Rotation

Double Rotation is performed to fix case 2 and case 3.

Case 2 :

An insertion into the right subtree of the left child.

General Representation

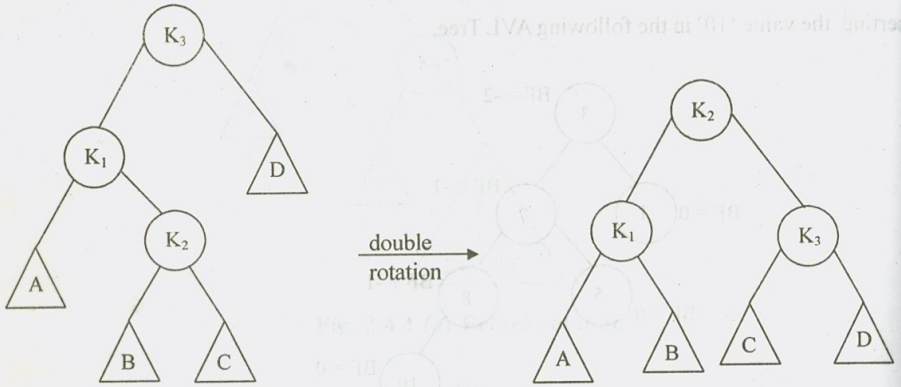
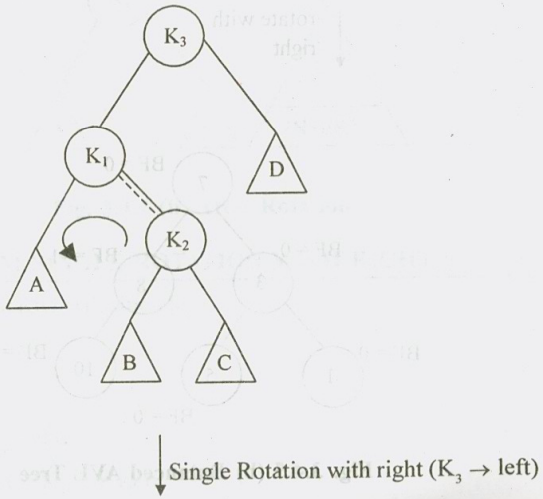


Fig. 3.4.6

This can be performed by 2 single rotations.



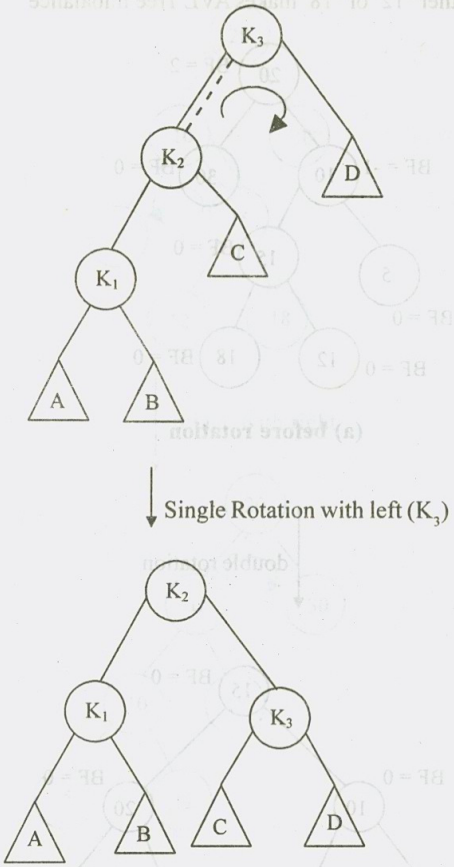
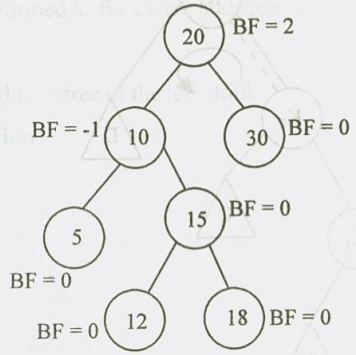


Fig. 3.4.7 Balanced AVL Tree

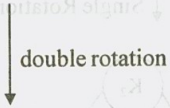
ROUTINE TO PERFORM DOUBLE ROTATION WITH LEFT :

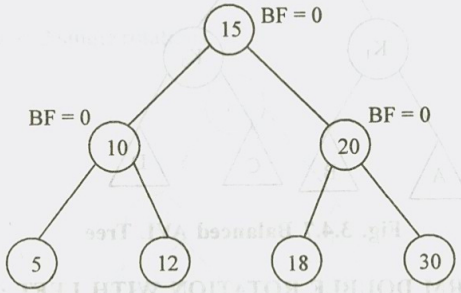
```
Double Rotate with left (Position K3)
{       /* Rotation Between K1 & K2 */
        K3 → Left = Single Rotate with Right (K3 → Left);
                /* Rotation Between K3 & K2 */
        Return Single Rotate With Left (K3);
}
```

Example : Insertion of either '12' or '18' makes AVL Tree imbalance



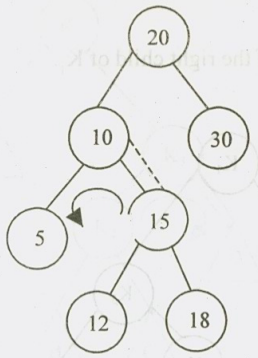
(a) before rotation

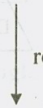


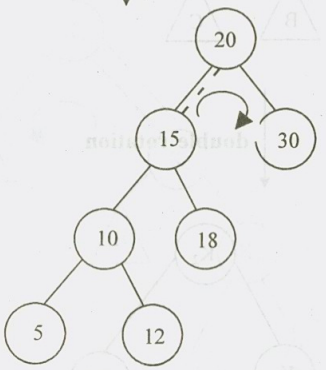


(b) After rotation

This can be done by performing single rotation with right of '10' & then perform the single rotation with left of 20 as shown below.







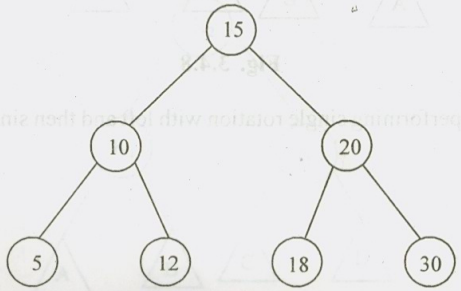


Fig. 3.4.7 Balanced AVL Tree

Case 4 :

An Insertion into the left subtree of the right child of K_1.

General Representation :-

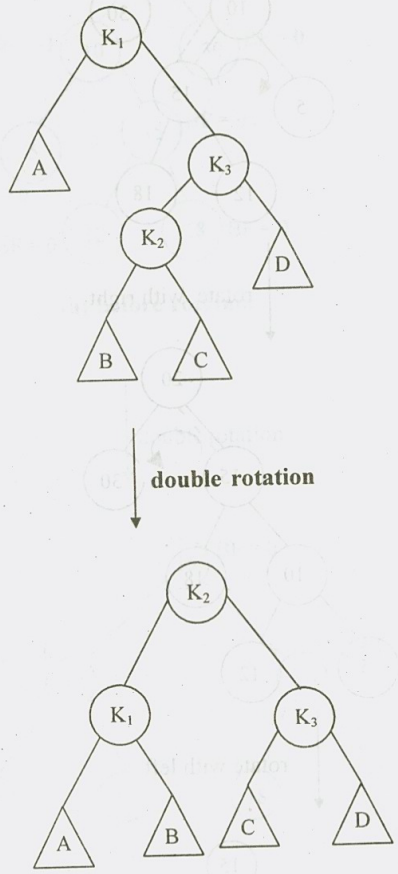


Fig. 3.4.8

This can also be done by performing single rotation with left and then single rotation with right.

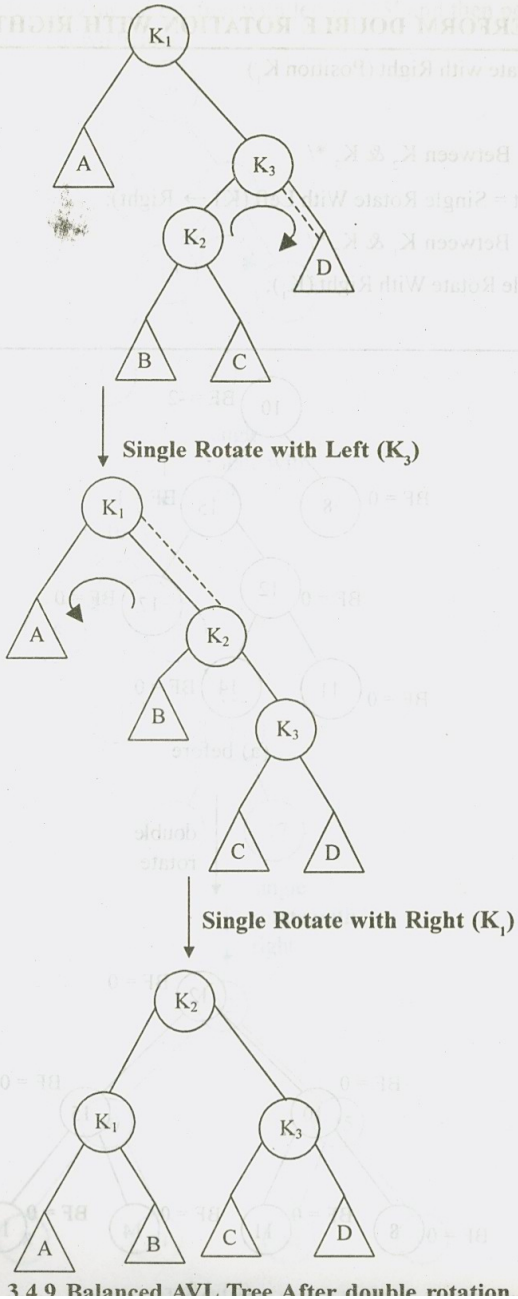
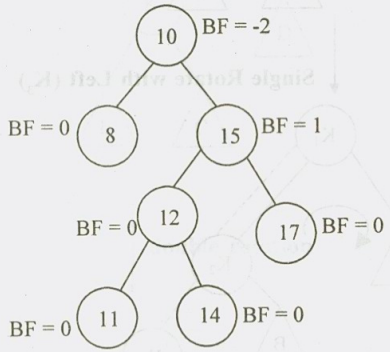


Fig. 3.4.9 Balanced AVL Tree After double rotation

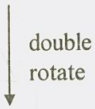
ROUTINE TO PERFORM DOUBLE ROTATION WITH RIGHT :

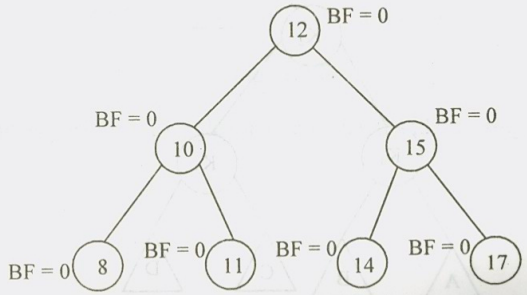
```
Double Rotate with Right (Position K1)
{
/* Rotation Between K2 & K3 */
K1 → Right = Single Rotate With Left (K1 → Right);
/* Rotation Between K1 & K2 */
return Single Rotate With Right (K1);
}
```

Example :



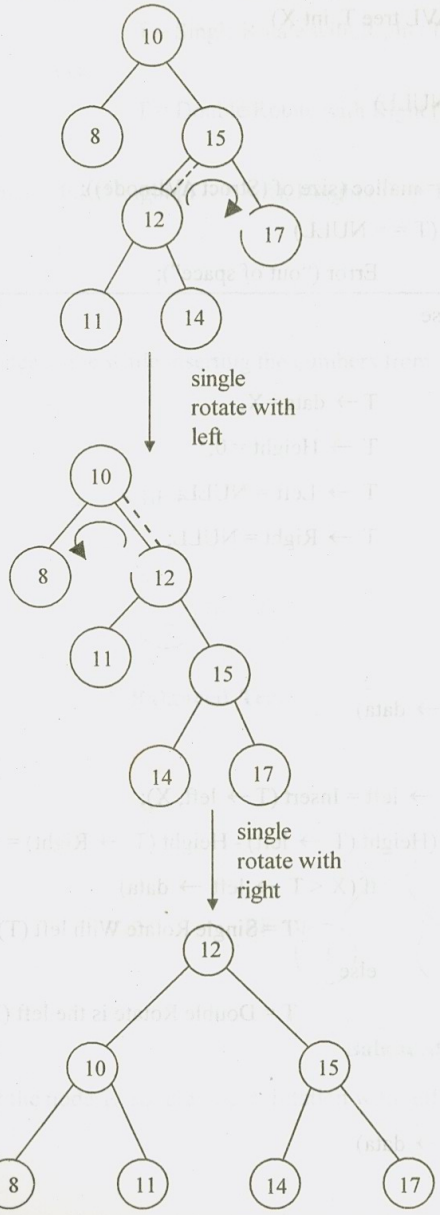
(a) before





(b) After

This can be done by performing single rotation with left of '15' and then performing the single rotation with right of '10' as shown below.



ROUTINE TO INSERT IN AN AVL TREE : -

```
AVLTree Insert (AVL tree T, int X)
{
    if (T == NULL)
    {
        T = malloc (size of (Struct AVLnode));
        if (T = = NULL)
            Error ("out of space");
        else
        {
            T → data =X;
            T → Height = 0;
            T → Left = NULL;
            T → Right = NULL;
        }
    }
    else
    if (X < T → data)
    {
        T → left = Insert (T → left, X);
        if (Height (T → left) - Height (T → Right) = = 2)
            if (X < T → left → data)
                T = Single Rotate With left (T);
            else
                T = Double Rotate is the left (T);
    }
    else
    if (X > T → data)
    {
        T → Right = Insert (T → Right, X);
        if (Height (T → Right) - Height (T → left) = = 2)
            if(X > T → Right → Element)
                T = Single Rotate with Right (T);
            else
                T = Double Rotate with Right (T);
    }
    T → Height = Max (Height (T → left), Height (T → Right))+1;
    return T;
}
```

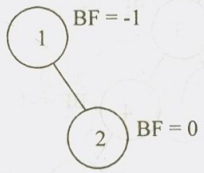
Example :

Let us consider how to balance a tree while inserting the numbers from 1 to 10.

Insert the value 1.

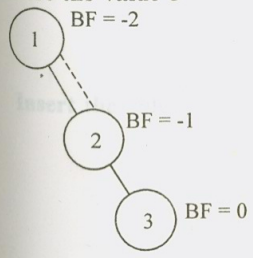
(1) BF = 0

Insert the value 2



Balanced Tree

Insert the value 3

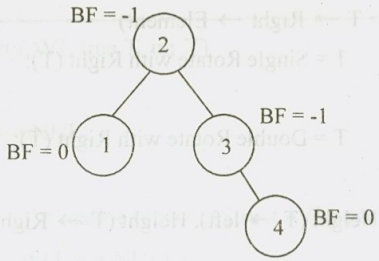


Imbalanced Tree

Balanced Tree

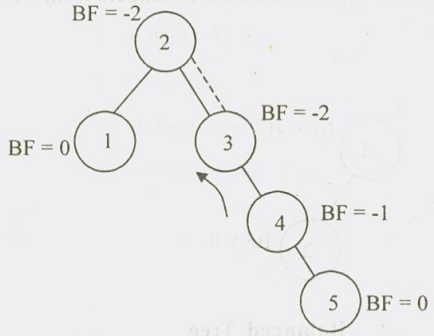
Here the tree imbalances at the node 1. so the single rotation with left is performed.

Insert the value 4

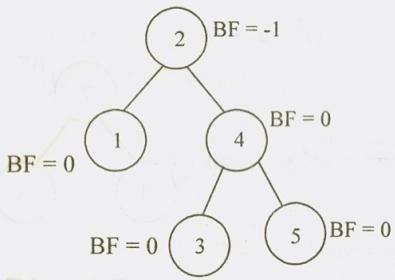


Balanced AVL Tree

Insert the value 5



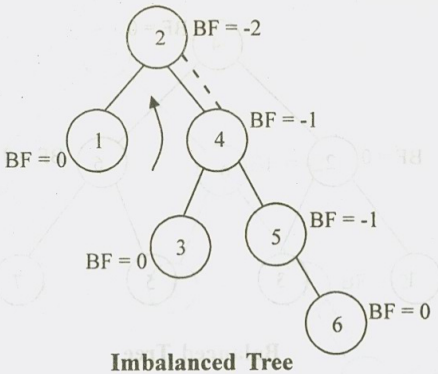
Imbalanced Tree



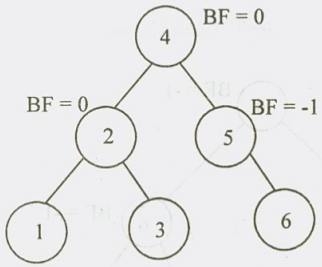
Balanced Tree

Tree is imbalanced at node '3', perform the single rotation with left to balance it.

Insert the value 6

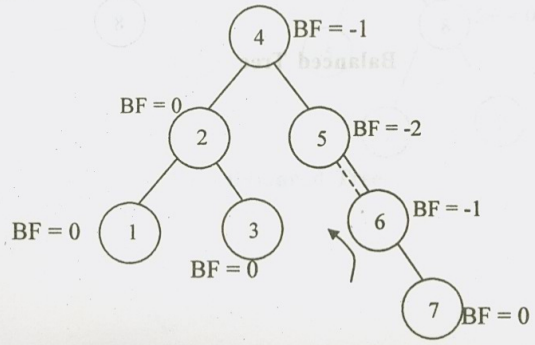


Imbalanced Tree

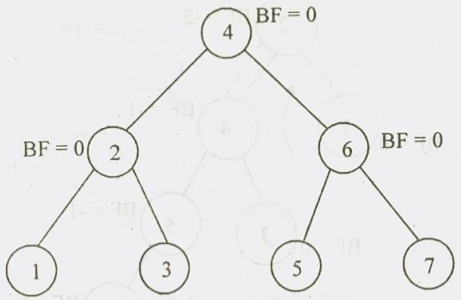


Balanced Tree

Insert the value 7

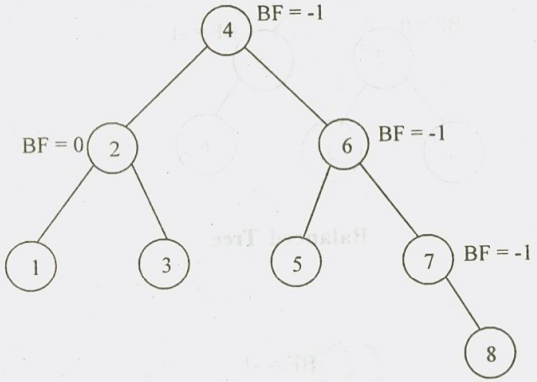


Imbalanced Tree



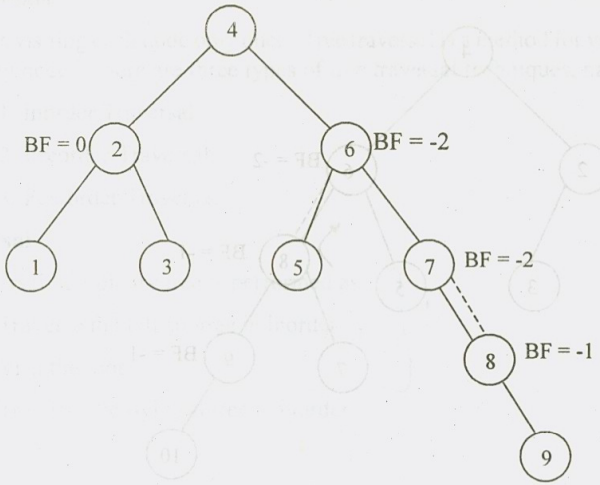
Balanced Tree

Insert the value 8

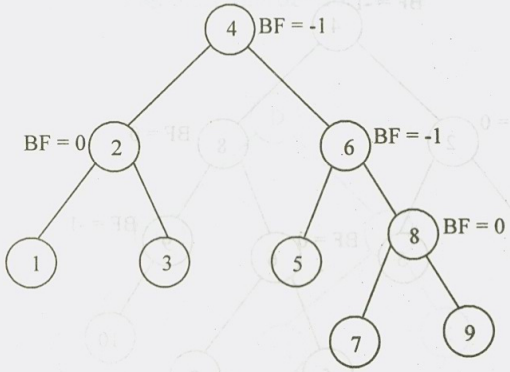


Balanced Tree

insert the value 9

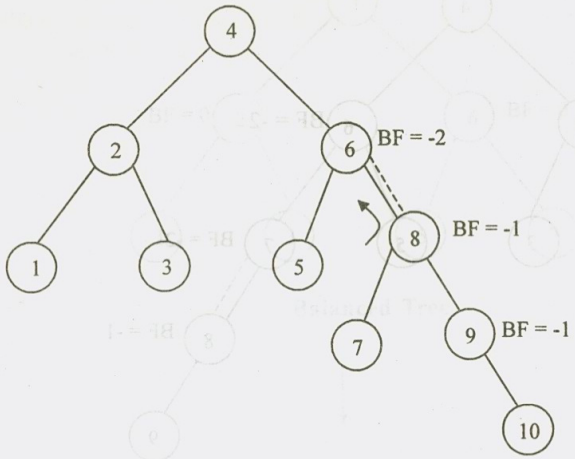


Imbalanced Tree

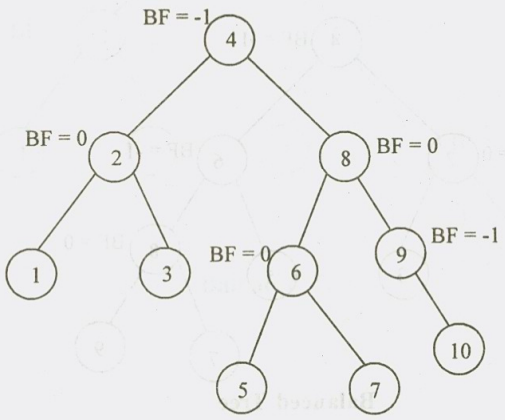


Balanced Tree

Insert the value 10



Imbalanced Tree



Balanced Tree

3.5 Tree Traversals

Traversing means visiting each node only once. Tree traversal is a method for visiting all the nodes in the tree exactly once. There are three types of tree traversal techniques, namely

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder Traversal

The inorder traversal of a binary tree is performed as

* Traverse the left subtree in inorder
* Visit the root
* Traverse the right subtree in inorder.

Example :

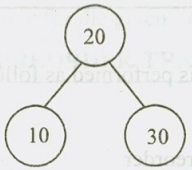


Fig. 3.5.1 Inorder 10, 20, 30

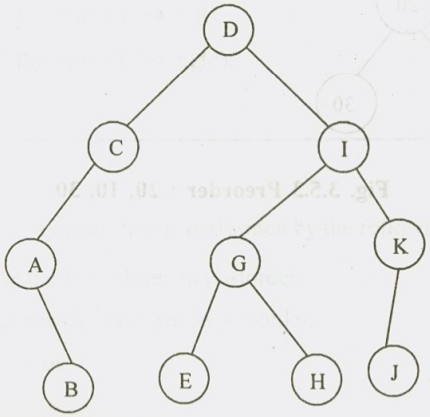


Fig. 3.5.2 A B C D E G H I J K

The inorder traversal of the binary tree for an arithmetic expression gives the expression in an infix form.

RECURSIVE ROUTINE FOR INORDER TRAVERSAL

```
void Inorder (Tree T)
{
    if (T ! = NULL)
    {
        Inorder (T → left);
        printElement (T → Element);
        Inorder (T → right);
    }
}
```

Preorder Traversal

The preorder traversal of a binary tree is performed as follows,

* Visit the root
* Traverse the left subtree in preorder
* Traverse the right subtree in preorder.

Example 1 :

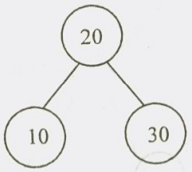


Fig. 3.5.3 Preorder : 20, 10, 30

Example 2 :

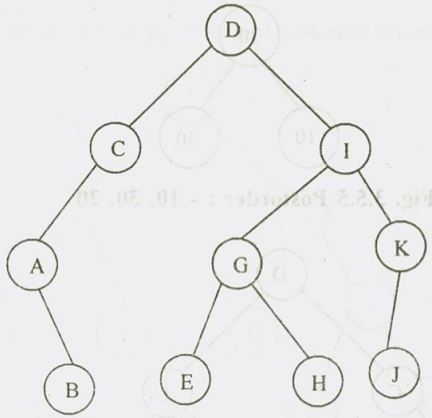


Fig. 3.5.4 Preorder D C A B I G E H K J

the preorder traversal of the binary tree for the given expression gives in prefix form.

RECURSIVE ROUTINE FOR PREORDER TRAVERSAL

```
void Preorder (Tree T)
{
    if (T ! = NULL)
    {
        printElement (T → Element);
        Preorder (T → left);
        Preorder (T → right);
    }
}
```

Postorder Traversal

The postorder traversal of a binary tree is performed by the following steps.

* Traverse the left subtree in postorder.
* Traverse the right subtree in postorder.
* Visit the root.

Example : 1

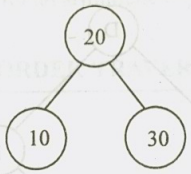


Fig. 3.5.5 Postorder : - 10, 30, 20

Example : 2

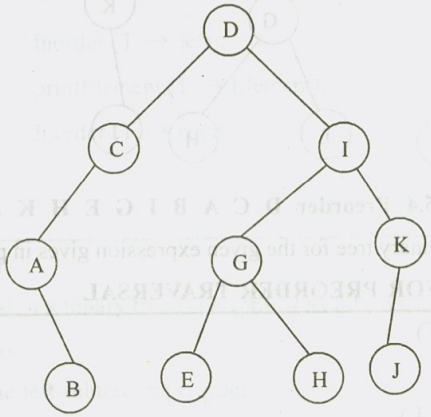


Fig. 3.5.6 Post order : - B A C E H G J K I D

The postorder traversal of the binary tree for the given expression gives in postfix form.

RECURSIVE ROUTINE FOR POSTORDER TRAVERSAL

```
void Postorder (Tree T)
{
    if (T ! = NULL)
    {
        Postorder (T → Left);
        Postorder (T → Right);
        PrintElement (T → Element);
    }
```

Example : -

Traverse the given tree using inorder, preorder and postorder traversals.

(1)

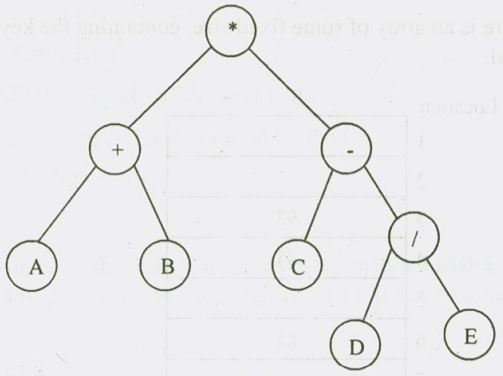


Fig. 3.5.7

Inorder : A + B * C - D / E

Preorder : * + A B - C / D E

Postorder : A B + C D E / - *

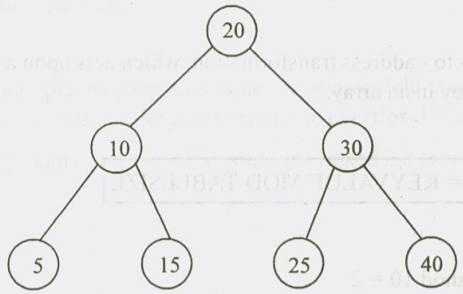


Fig. 3.5.8

Inorder : 5 10 15 20 25 30 40

Preorder : 20 10 5 15 30 25 40

Postorder : 5 15 10 25 40 30 20

3.6 Hashing

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

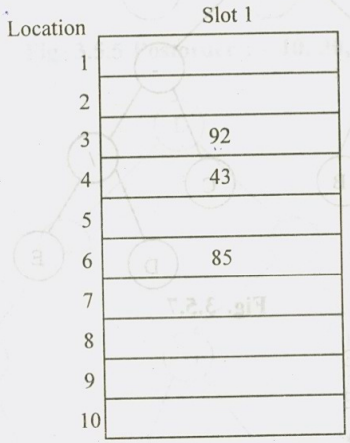


Fig. 3.6.1 Hash Table

Hashing Function

A hashing function is a key - to - address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example : - Hash (92)

Hash (92) = 92 mod 10 = 2

The keyvalue '92' is placed in the relative location '2'.

ROUTINE FOR SIMPLE HASH FUNCTION

```
Hash (Char *key, int Table Size)
{
        int Hashvalue = 0;
        while (* key ! = '\0')
                Hashval + = * key ++;
        return Hashval % Tablesize;
```

Some of the Methods of Hashing Function

1. Module Division
2. Mid - Square Method
3. Folding Method
4. PSEUDO Random Method
5. Digit or Character Extraction Method
6. Radix Transformation.

Collisions

Collision occurs when a hash value of a record being inserted hashes to an address (i.e. Relative position) that already contain a different record. (ie) When two key values hash to the same position.

Collision Resolution

The process of finding another position for the collide record.

Some of the Collision Resolution Techniques

1. Seperate Chaining
2. Open Addressing
3. Multiple Hashing

Seperate Chaining

Seperate chaining is an open hashing technique. A pointer field is added to each record location. When an overflow occurs this pointer is set to point to overflow blocks making a linked list.

In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insert : 10, 11, 81, 10, 7, 34, 94, 17

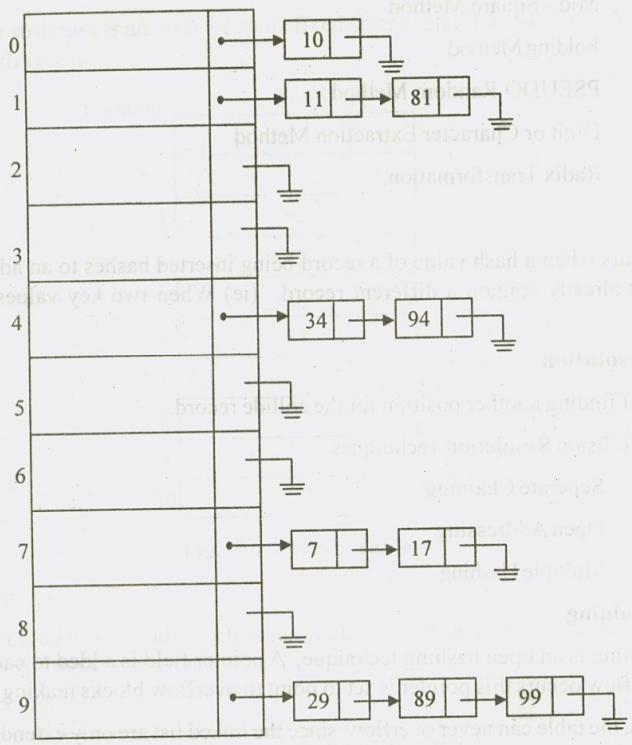


Fig. 3.6.2

Insertion

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the element turns to be a new one, it is insertd either at the front of the list or at the end of the list.

If it is a duplicate element, an extra field is kept and placed.

INSERT 10 :

Hash (k) = k% Tablesize

Hash (10) = 10 % 10

Hash (10) = 0

INSERT 11 :

Hash (11) = 11 % 10

Hash (11) = 1

INSERT 81 :

Hash (81) = 81% 10

Hash (81) = 1

The element 81 collides to the same hash value 1. To place the value 81 at this position perform the following.

Traverse the list to check whether it is already present.

Since it is not already present, insert at end of the list. Similarly the rest of the elements are inserted.

ROUTINE TO PERFORM INSERTION

```
void Insert (int key, Hashtable H)
{
        Position Pos, Newcell;
        List L;
/* Traverse the list to check whether the key is already present */
        Pos = FIND (Key, H);
        If (Pos = = NULL) /* Key is not found */
        {
                Newcell = malloc (size of (struct ListNode));
                If (Newcell ! = NULL)
                (
                   L = H → Thelists [Hash (key, H→ Tablesize)];
                   Newcell → Next = L → Next;
                   Newcell → Element = key;
                   /* Insert the key at the front of the list */
                    L → Next = Newcell;
                }
        }
}
```

FIND ROUTINE

```
Position Find (int key, Hashtable H)
{
        Position P;
        List L;
        L = H → Thelists [Hash (key, H → Tablesize)];
        P = L → Next;
        while (P! = NULL && P → Element ! = key)
                P = p → Next;
        return p;
}
```

Advantage

More number of elements can be inserted as it uses array of linked lists.

Disadvantage of Seperate Chaining

* It requires pointers, which occupies more memory space.
* It takes more effort to perform a search, since it takes time to evaluate the hash function and also to traverse the list.

OPEN ADDRESSING

Open addressing is also called **closed Hashing**, which is an alternative to resolve the collisions with linked lists.

In this hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. (ie) cells $h_0(x)$, $h_1(x)$, $h_2(x)$.....are tried in succession.

There are three common collision resolution strategies. They are

(i) Linear Probing

(ii) Quadratic probing

(iii) Double Hashing.

LINEAR PROBING

In linear probing, for the i^{th} probe the position to be tried is (h(k) + i) mod tablesize, where F(i) = i, is the linear function.

In linear probing, the position in which a key can be stored is found by sequentially searching all position starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cells has been found, then the search is continued

INSERT : 42, 39, 69, 21, 71, 55, 33

	EMPTY TABLE	After 42	After 39	After 69	After 21	After 71	After 55	After 33
0				69	69	69	69	69
1					21	21	21	21
2		42	42	42	42	42	42	42
3						71	71	71
4								33
5							55	55
6								
7								
8								
9			39	39	39	39	39	39

In fig 3.6.3 first collision occurs when 69 is inserted, which is placed in the next available spot namely spot 0, which is open.

The next collision occurs when 71 is inserted, which is placed in the next available spot namely spot 3. The collision for 33 is handled in a similar manner.

Advantage :

* It doesn't requires pointers

Disadvantage

* It forms clusters, which degrades the performance of the hash table for storing and retrieving data.

3.7 PRIORITY QUEUES

3.7.1 Need for Priority Queue

In multiuser environment, the operating system scheduler must decide which of several processes to run. Generally a process is allowed to run only for a fixed period of time. If an algorithm uses simple queue, jobs are initially placed at the end of the queue. The scheduler will take the first job on the queue and allow it to run until either it finishes or its timelimit is up. If it is not completed the alloted time, then place it at the end of the queue.

This is not appropriate, because very short jobs will take a long time because of the wait involved to run. It can be overcome by using priority queues in which short jobs are assigned a higher precedence. Futhermore, some jobs that are not short are still very important and should also have preferences.

3.7.2 Model

A priority queue is a special kind of queue data structure which will have precedence over jobs.

Basic operations performed by priority queue are :

Insert Operation

DeleteMin Operation

Insert operation is similar to Enqueue, where we insert an element in the priority queue.

Deletemin operation is equivalent of queue's Dequeue operations, where we delete at minimum element from the priority queue.

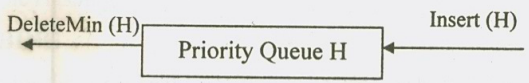


Fig. 3.7.1 Basic Model of a priority Queue

3.7.3 Simple Implementation

There are several ways for implementing priority queue.

1. Linked list
2. Binary search tree
3. Binary heap

Linked List

A simple linked list implementation of priority queue requires O(1) time to perform the insertion at the front and O(N) time to delete the minimum element.

Binary Search Tree

This implementation gives an average running time of O(logN) for both insertion and deletemin operations.

Deletemin operation makes tree imbalance, by making the right subtree heavy as we are repeatedly removing the node from the left subtree.

3.7.4 BINARY HEAP

The efficient way of implementing priority queue is Binary Heap. Binary heap is merely referred as Heaps, Heap have two properties namely

* Structure property
* Heap order property.

Like AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. Both the operations require the average running time as O(log N).

Structure Property

A heap should be complete binary tree, which is a completely filled binary tree with the possible exception of the bottom level, which is filled from left to right.

A complete binary tree of height H has between 2^H and 2^{H+1} -1 nodes.

For example if the height is 3. Then the numer of nodes will be between 8 and 15. (ie) (2^3 and 2^4-1).

For any element in array position i, the left child is in position 2i, the right child is in position 2i + 1, and the parent is in i/2. As it is represented as array it doesn't require pointers and also the operations required to traverse the tree are extremely simple and fast. But the only disadvantage is to specify the maximum heap size in advance.

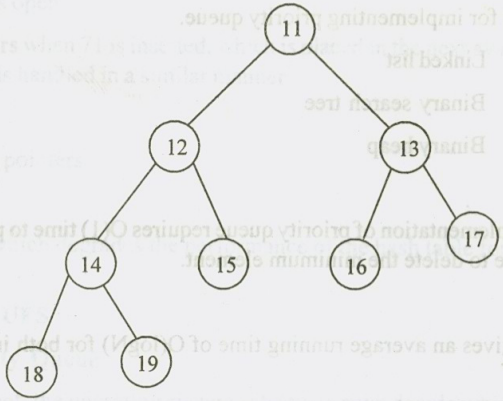


Fig. 3.7.2 A Complete Binary Tree

	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Fig. 3.7.3 Array implementation of complete binary tree

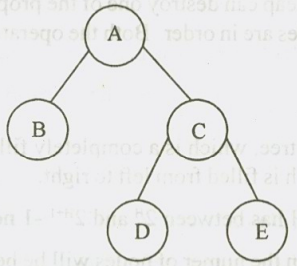


Fig. 3.7.4 Not a Complete Binary Tree

Heap Order Property

In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of the root (which has no parent).

This property allows the deletemin operations to be performed quickly has the minimum element can always be found at the root. Thus, we get the FindMin operation in constant time.

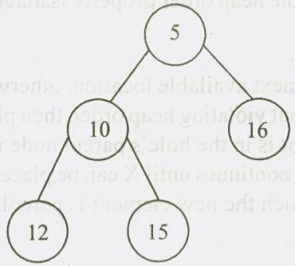


Fig. 3.7.5 (a) Binary tree with structure and heap order property.

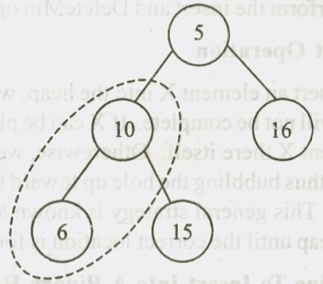


Fig. 3.7.5 (b) Binary tree with structure but violtating heap heap order property

Declaration for priority queue

```
Struct Heapstruct;
typedef struct Heapstruct * priority queue;
PriorityQueue Initialize (int MaxElements);
void insert (int X, PriorityQueue H);
int DeleteMin (PriorityQueue H);
Struct Heapstruct
{
        int capacity;
        int size;
        int *Elements;
};
```

Initialization

```
PriorityQueue Initialize (int MaxElements)
{
        PriorityQueue H;
        H = malloc (sizeof (Struct Heapstruct));
        H→ Capacity = MaxElements;
        H→ size = 0;
        H → elements [0] = MinData;
        return H;
```

3.6.4.3 BASIC HEAP OPERATIONS

To perform the insert and DeleteMin operations ensure that the heap order property is maintained.

Insert Operation

To insert an element X into the heap, we create a hole in the next available location, otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then place the element X there itself. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. This process continues until X can be placed in the hole. This general strategy is known as Percolate up, in which the new element is percolated up the heap until the correct location is found.

Routine To Insert Into A Binary Heap

```
void insert (int X, PriorityQueue H)
{
        int i;
        If (Isfull (H))
        {
                Error (" priority queue is full");
                return;
        }
        for (i = ++H → size; H → Elements [i/2] > X; i/=2)
        /* If the parent value is greater than X, then place the element of parent
                node into the hole */.
                H → Elements [i] = H → Elements [i/2];
                H → elements [i] = X; // otherwise, place it in the hole.
}
```

Example :

To Insert 10 :

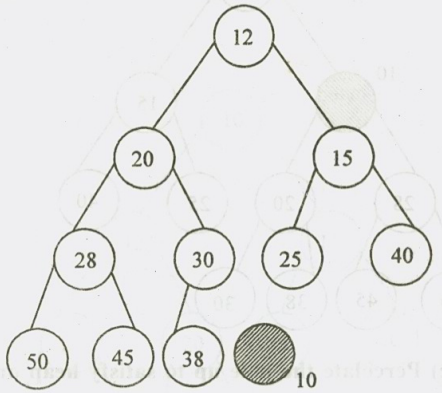


Fig. 3.7.6(a) A hole is created at the next location

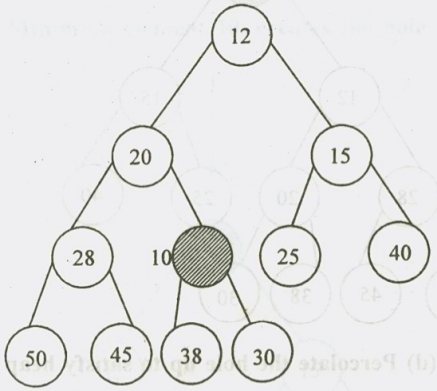


Fig. 3.7.6 (b) Percolate the hole up to satisfy heap order

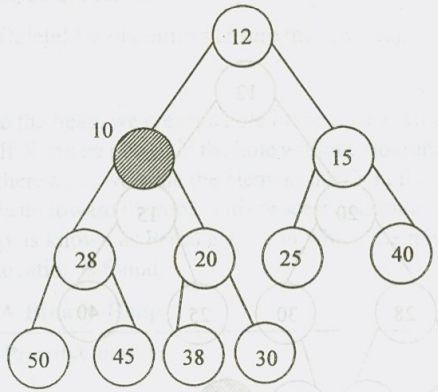


Fig. 3.7.6 (c) Percolate the hole up to satisfy heap order

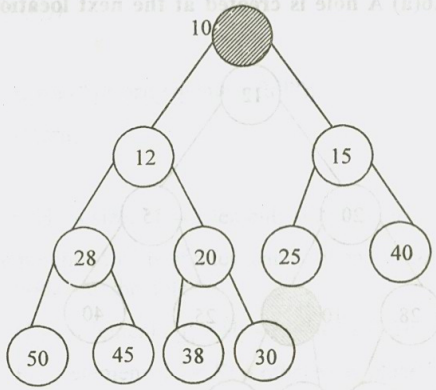


Fig. 3.7.6 (d) Percolate the hole up to satisfy heap order

In Fig 3.6.6(d) the value 10 is placed in its correct location.

DeleteMin

DeleteMin Operation is deleting the minimum element from the Heap.

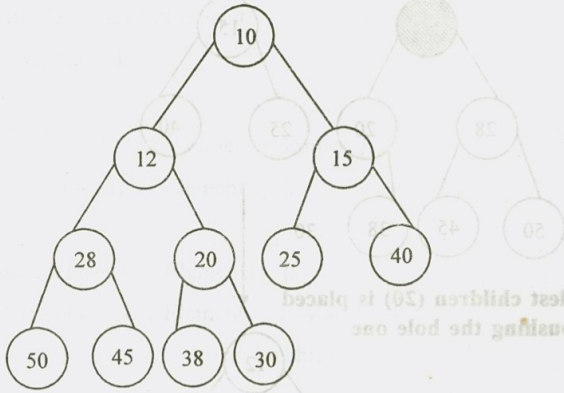
In Binary heap the minimum element is found in the root. When this minimum is removed, a hole is created at the root. Since the heap becomes one smaller, makes the last element X in the heap to move somewhere in the heap.

If X can be placed in hole without violating heaporder property place it.

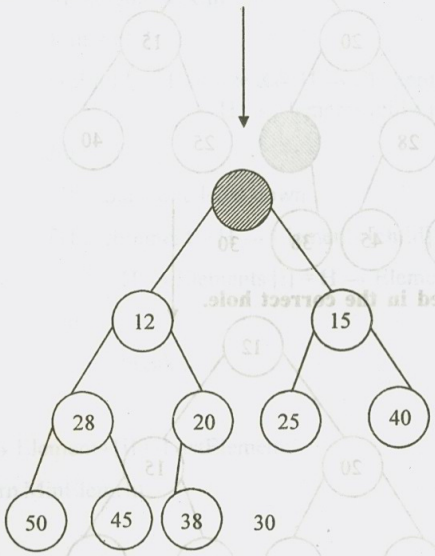
Otherwise, we slide the smaller of the hole's children into the hole, thus pushing the hole down one level.

We repeat until X can be placed in the hole. This general strategy is known as perculate down.

Example: To delete the minimum element 10

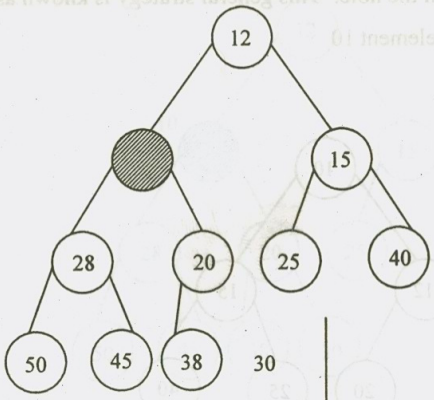


Delete Minimum element 10, creates the hole at the root.

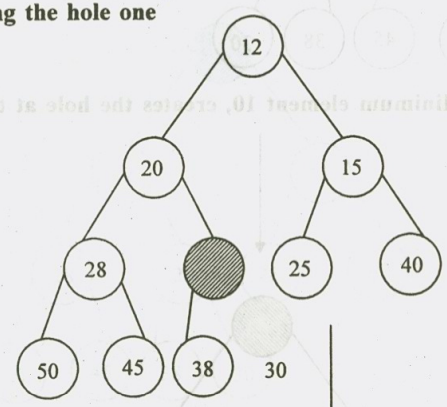


The last element '30' must be moved somewhere in the heap.

The Hole's smallest children (12) is placed into the hole by pushing the hole down one level.



The hole's smallest children (20) is placed into the hole by pushing the hole one level down.



The last element '30' is placed in the correct hole.

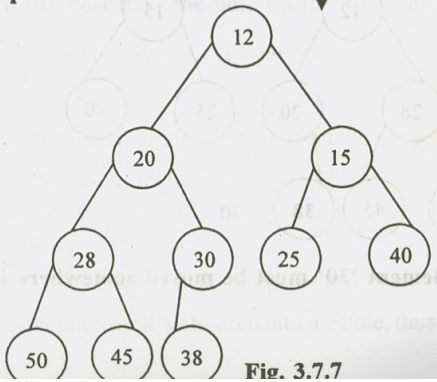


Fig. 3.7.7

ROUTINE TO PERFORM DELETEMIN IN A BINARY HEAP

```
int Deletemin (PriorityQueue H)
{
    int i, child;
    int MinElement, LastElement;
    if (IsEmpty (H))
    {
        Error ("Priority queue is Empty");
        return H → Elements [0];
    }
    MinElement = H → Elements [1];
    LastElement = H→ Elements [H → size - -];
    for (i = 1; i * 2 < = H → size; i = child)
    {
        /* Find Smaller Child */
        child = i * 2;
        if (child ! = H → size && H → Elements [child + 1]
                < H → Elements [child])
        child + +;
        // Percolate one level down
        if (LastElement > H → Elements [child])
            H → Elements [i] = H → Elements [child];
        else
            break ;
    }
    H → Elements [i] = LastElement;
    return MinElement;
}
```

OTHER HEAP OPERATIONS

The other heap operations are

(i) Decrease - key

(ii) Increase - key

(iii) Delete

(iv) Build Heap

DECREASE KEY

The Decreasekey (P, Δ, H) operation decreases the value of the key at position P by a positive amount Δ. This may violate the heap order property, which can be fixed by percolate up.

example : Priority Queue H

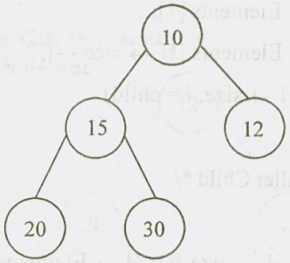
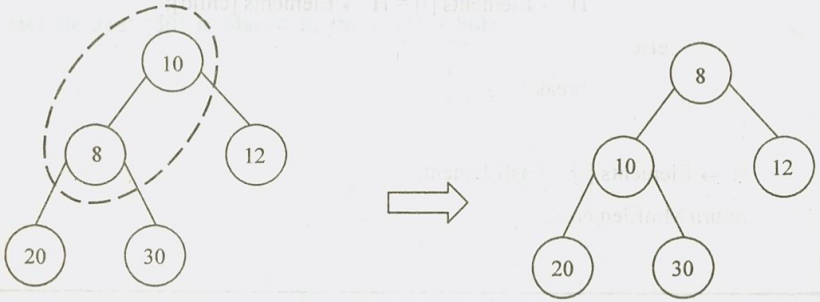


Fig. 3.7.8 (a) Decrease Key (2, 7, H)

Element at position 2 is '15'. Decrease that element by 7. Now the position 2 has the value '8', which violates the heap order property.

This can be fixed by percolating up strategy.

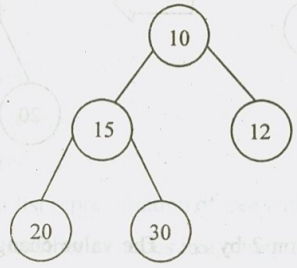


Increase - Key

The increase - key (p, Δ, H) operation increases the value of the key at position p by a positive amount Δ. This may violate heap order property, which can be fixed by percolate down.

example :

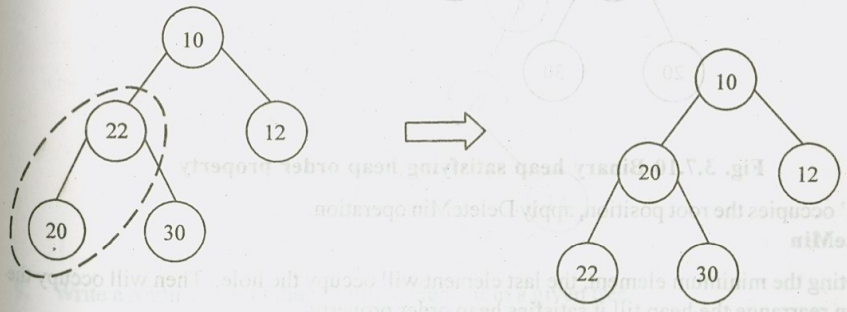
Priority Queue H



Increase Key (2, 7, H)Fig. 3.7.9 (a)

Here, the Element at position 2 is 15. Increase that value by 7. Now the position 2 has the value 22, which violates the heap order property.

This can be fixed by percolate down.



Delete :

The Delete (P, H) operation removes the node at the position P from the heap H. This can be done by.

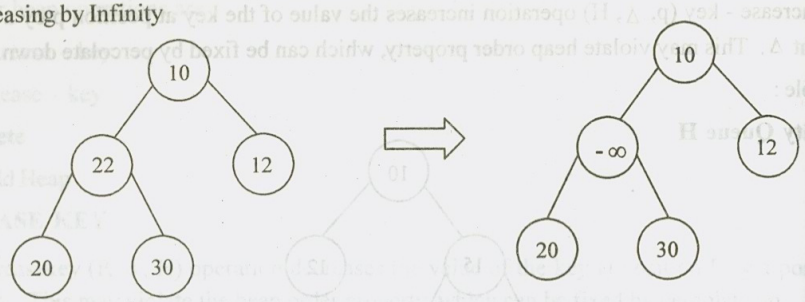
(i) Perform the decreasekey operation

Decreasekey (P, ∞, H)

(ii) Perform Deletemin operation

eg : - Delete (2, ∞, H)

(i) Decreasing by Infinity



After Decreasing the value at position 2 by ∞. The value changes to -∞, which is the least element in heap.

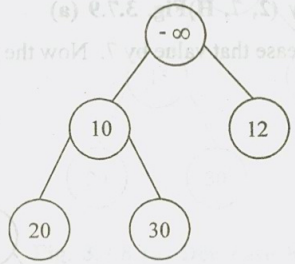
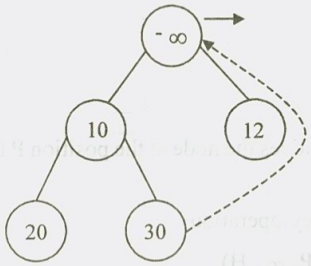


Fig. 3.7.10 Binary heap satisfying heap order property

since '-∞' occupies the root position, apply DeleteMin operation.

(ii) DeleteMin

After deleting the minimum element, the last element will occupy the hole. Then will occupy the hole. Then rearrange the heap till it satisfies heap order property.



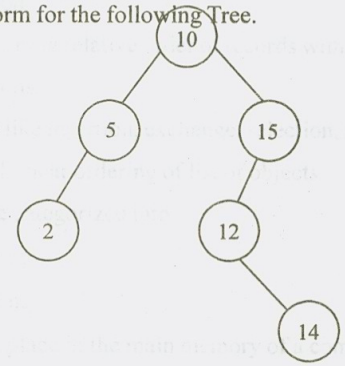
Build Heap

The Build Heap (H) operations takes as input N keys and places them into an empty heap by

Questions

Part - A

1. Compare General Tree and binary tree?
2. Define the following terminologies in a tree
   (1) Siblings, parent
   (2) Depth, Path
   (3) Height, Degree
3. What is complete binary tree?
4. Define Binary Search Tree.
5. Give the array and linked list representation of tree with an example.
6. Show that the maximum number of nodes in a binary tree of height H as 2^{H+1} -1.
7. Define Tree Traversal.
8. Give the preorder form for the following Tree.



9. Write a routine to find the minimum element in a given tree.
10. Write the recursive procedure for inorder traversals.
11. Draw a binary search tree for the following input lists. 60, 25, 75, 15, 33, 44
2. How is a binary tree represented using an array?
13. Define AVL tree.
14. What are the two properties of a binary heap.
15. Define (i) Hashing (ii) Collision (iii) Hash function.
16. What is open addressing?
17. Write a routine to perform single rotate with left.